SOFTWARE PROCESS IMPROVEMENT AND PRACTICE
Softw. Process Improve. Pract. 2005; **10**: 103–124
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spip.222

Four Mechanisms for Adaptable Systems: A Meta-level Approach to Building a Software Product Line

Research Section

Claudia Fritsch[1]*and Burkhardt Renz[2]
[1] *Robert Bosch GmbH, P.O. Box 94 03 50, D-60461 Frankfurt, Germany*
[2] *University of Applied Sciences Giessen-Friedberg, Department MNI, Wiesenstr. 14, D-35390 Giessen, Germany*

Meta-level architectures combined with domain-specific languages serve as a powerful tool to build and maintain a software product line: Meta-level architectures lead to adaptable software systems. Executable descriptions capture expert knowledge.

We have developed a meta-level architecture for a software product line of legal expert systems. Four meta-level mechanisms support both variability and evolution of the product line. Domain analysis had shown that separation of expert knowledge from technical code was essential. Descriptions written in domain-specific languages reside in the meta level, and serve as specification, code, and documentation. Technical code finds its place in interpreting machines in the base level.

We discuss how meta-level architectures influence the qualities of software product lines and how properties and patterns of the problem space can guide the design of domain-specific languages. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: software product lines; software architecture; meta-level architectures; domain-specific languages; adaptable software

1. INTRODUCTION

For more than 10 years, we have developed and maintained a software product line of legal expert systems. This software product line is made by a German publishing house specializing in international civil law and law of civil status. Today, the company is the German market leader. On the basis of our experience, we discuss the influence of meta-level architectures on product line engineering, and the advantages of domain-specific languages.

The products of this product line share certain functionality, such as interaction with the user by means of a graphical interface, filing data, storing information in a database, and printing documents. They differ mainly in two points:

- *Domain descriptions* defining input and output as well as how to produce output from input. These descriptions depend on complex legal regulations.

* Correspondence to: Claudia Fritsch, Robert Bosch GmbH, Corporate Research and Development, P.O. Box 94 03 50, D-60461 Frankfurt, Germany
† E-mail: Claudia.Fritsch@de.bosch.com

- *Technical infrastructure* comprising operating system, GUI framework, and database system. Different constraints emerge through changes both in customer requirements and in technology over time.

Section 2 introduces the product line.

When we designed the architecture for this software product line, we focused on two requirements in particular:

- Domain experts should be involved in development, but should not have to learn a general-purpose programming language.
- Changes in domain descriptions should leave technical code untouched and vice versa.

The solution for both was to separate the programs' domain descriptions from technical code. Using a *meta-level architecture*, we achieved a sound decoupling: Domain descriptions written in *domain-specific languages* are kept in the meta level. Appropriate engines included in the base level act according to these descriptions. Section 3 explains this architecture.

In Section 4, we present the four meta-level *mechanisms* that we have developed for the design of this software product line. They separate domain descriptions from technical code in the following areas:

- data access
- filing input and printing output
- application logic
- program control flow.

These mechanisms have brought many benefits:

- Highly valuable domain expertise is captured and is readable, maintainable, and available for developers. It is also guaranteed to be up-to-date.
- The software is quickly adaptable to changes in the domain or in technology.
- Domain descriptions are explicitly present in the meta level, executed by the engines in the base level. Traceability is therefore trivial.

Section 5 discusses the interesting effects of a meta-level architecture on product line engineering. Our architecture and the languages are used in all products. Variability of the domain resides in the meta level, variability of technology in the base level. We discuss the impact on quality attributes, namely modifiability, performance, testability, availability, security, and usability. Our architecture enhances modifiability and usability, which both were major architectural goals.

The domain-specific languages that we developed play a major role in the success of our product line, its efficient development, and the capturing of domain knowledge. However, these benefits are not guaranteed. The scope of these languages defines the capabilities of the products. One has to analyze the domain systematically and design the languages thoroughly. Section 6 emphasizes the indispensability of understanding the domain and gives guidelines for the design of domain-specific languages.

Finally, Section 7 gives a conclusion.

2. A PRODUCT LINE OF LEGAL EXPERT SYSTEMS

The products in this software product line are made for registrars and similar offices. The product line covers the German and Austrian market. Variants reflect national and regional legislation and practice.

The software processes *civil transactions*, such as the registration of births or marriages. Inputs for each civil transaction are *personal data* that are processed according to complex legal rules. The output of each civil transaction is a set of *documents*. Customers demand software guaranteed to be legally correct (i.e. observing the Law on Personal Status (Schmitz and Bornhofen 2003)).

2.1. Scope

German and Austrian laws of civil status have the same structure (as distinct from the Anglo-Saxon juridical system), but differ in many details. Austrian registrars additionally administer citizenship – a transaction that does not belong to the duty of German registrars. In Germany, Land law (state law) adds specific rules to federal law. This demands variants of the software. The government has assigned special tasks to some registry offices, requiring more variants. While these variations result from differences in the domain, other reasons for variability stem from technology.

The products have to support different technologies demanded by the customers. Registry offices are equipped with different hardware and software: some have networks, others do not. Some use

relational database management systems, such as Oracle or MS SQL Server, others ask for a low-cost (or even no-cost) database. We call combinations of these technologies *platforms.* The use of these different platforms has consequences for the software. Nevertheless, the software has to offer the same, required functionality on each platform.

Some products in this product line have been on the market for more than 10 years. Technology has been changing; accordingly, the software has been subject to change. Customers follow the technology change. Some of them do it fast, but the financial situation of others does not allow up-to-date technology (e.g. the switch to 32-Bit-Windows spread over more than 5 years). This is another reason why the product line has to support different platforms at the same time: The products' life cycles overlap.

Changes in the domain are initiated by changes in the law or the development of society reflected in the law, and they occur independently of changes in technology. Technical changes and domain-specific changes have to be made at the same time, but without interference.

2.2. Basic Product for German Registrars

To illustrate the characteristics of the domain, we give an overview of the product for German registrars below.

A German registry office is divided into five *departments.* The work in each department is divided into 4–12 *categories*. Examples of departments are births, deaths, and marriages. Examples of categories in the birth department are registration of a birth, legitimation of a child, and adoption. Altogether, there are about 40 categories. In each category, *civil transactions* are processed. An example of a civil transaction is the registration of the birth of one child.

A civil transaction may come in many *varieties*, depending on the family background and number of people involved. The varieties of the transactions rank from simple standard to highly complex involving international law. So, even within the same category, the amount of data required for one civil transaction differs. It might involve a different number of people, different nationalities, and so on.

Many civil transactions are processed completely within an hour, others take weeks to be finalized. For example, the registrar may enter the data on a marriage up to 6 months in advance.

When a civil transaction has been processed, the output is a set of documents. *Forms*, required by law, must be filled out. Examples of documents are personal registration certificates, register entries, decrees, and notifications to other offices.

The output is strictly regulated by law (Schmitz and Bornhofen 2001, 2003), i.e. the layout of documents and even the wording of the contents. Sometimes, the rules are incomplete or conflicting. These cases are left to the registrar's discretion. The software, although an expert system, must leave final decisions to the registrar.

2.3. Domain Characteristics

The following characteristics are the key results of domain analysis.

- The civil transactions have the following properties in common:
  - A varying amount of data is required, depending on the background and number of people involved.
  - The required data and the documents that have to be printed follow certain patterns, depending on circumstances such as nationality or marital status.
  - A registrar may work on one civil transaction for a long time.
  - Data may be incomplete during processing.
  - Data may not be changed after registration.
- The law of civil status
  - may change several times a year, and time between proclamation and the effective date is short
  - can be mapped to processing rules to a high degree, but these rules may not limit the registrar's authority
  - is incorporated in forms, certificates, and a flow of work; registrars fill out forms and domain experts 'think' in forms, too.
- The registrars and personnel in registry offices
  - may either be experts in the field or have basic knowledge
  - have different computer skills
  - work in about 6000 German registry offices equipped with various platforms.

2.4. Requirements

From the domain characteristics, we derive the requirements our products have to fulfill. The following requirements lead to the four mechanisms we are going to describe:

- Usability: The software should
  - capture domain expertise and map it to processing rules
  - adapt to working methods in the office (not vice versa)
  - reflect the registrars' way of thinking, the work process, and the division of labor in the offices
  - support many varieties of a civil transaction, depending on the current data
  - guarantee legal correctness.
- Maintainability: The developers have to
  - adapt the software to changes of regulations several times a year
  - implement and test changes quickly
  - adapt the software to new technology without affecting the captured domain expertise and vice versa
- Platform independence: The software should
  - run on different operating systems
  - offer several databases
  - be easily adaptable to different user interfaces (GUIs)

3. ARCHITECTURE

The products in our product line share a common architecture. We outline this architecture, focusing on the points that give reasons for the four mechanisms.

3.1. Architectural Decisions

1. **Organize the processing of a civil transaction in a series of masks.** Users and domain experts are used to working with *forms.* Forms guide them through the processing of a civil transaction. The software should support this working method. Our basic decision was therefore to

- collect the data needed to process a civil transaction in a series of *masks* (data entry screens) that *act as a master form*
- use this data to print all documents (certificates, register entries, notifications, etc.)
- react on input immediately, guiding the user through the transaction

This decision implies the following:

- The series of masks depends highly on the current data of the concrete transaction. The data affects the number and order of masks presented and the control of the input on the masks.
- The masks contain domain knowledge, both legal regulations and common practice.

With this, the domain logic and the presentation of the application (in the form of series of masks) would be coupled tightly if we implemented the masks in a general-purpose language using the infrastructure of a specific GUI class library. Instead:

2. **Describe masks in a domain-specific language.** We designed a *domain-specific language* in which domain experts describe the layout and behavior of masks, in particular,

- input fields
- constraints on these fields
- control of input according to domain requirements.

This language does not depend on any GUI class library; rather, it yields a new level of abstraction. We regard this separating of the domain knowledge from the technical infrastructure as *decoupling by abstraction.* Masks are described without reference to a specific implementation, but are *translatable* into a representation for a certain GUI.

However, this second step in the development of the architecture has a drastic consequence: Naming data items and accessing their values have to be basic elements of the language.

3. **Reference data by symbolic names.** We abstract the actual storage of data to a data model where items are referenced by *symbolic names* (see mechanism #1). It turns out that this mechanism is the base of our architecture. The data items referenced by symbolic names form the *repository* that glues together

- the controlling of the masks
- the generation of documents
- the control flow of the application.

As a further result, data reference is database independent.

These ideas lead to the following principles, which we will use over and over again:

A meta-level architecture separates domain descriptions from technical code. A prerequisite is to find the *properties* of the processing of a civil transaction, both the content and the workflow. These properties are not part of the code, but allow us to describe the civil transaction at the meta-level. The base level executes these domain descriptions by means of interpreters and data-controlled engines, written in general-purpose languages (C, C++, Java). This permits us to move to another technology (e.g. a new database system or another operating system) without touching the expert knowledge.

Domain expertise is captured in executable descriptions. All expert knowledge is contained in *domain descriptions*, namely masks, documents, domain logic, and the data model. *Domain-specific languages* enable domain experts to describe their highly complex knowledge readable, maintainable, and executable: These descriptions control the base level. The captured knowledge is our most valuable core asset.

Domain descriptions are strictly separated from technical code and each description is in only one place. This allows us to embrace changes in the domain.

3.2. Overview of the Architecture

Figure 1 shows the high-level, compositional structure of our product line architecture[1] The components needed at runtime are shown on the left;

[1] As notation, we use Fundamental Modeling Concepts (FMC) developed by Siegfried Wendt and described in Knoepfel (2003) and Keller *et al*. (2002).

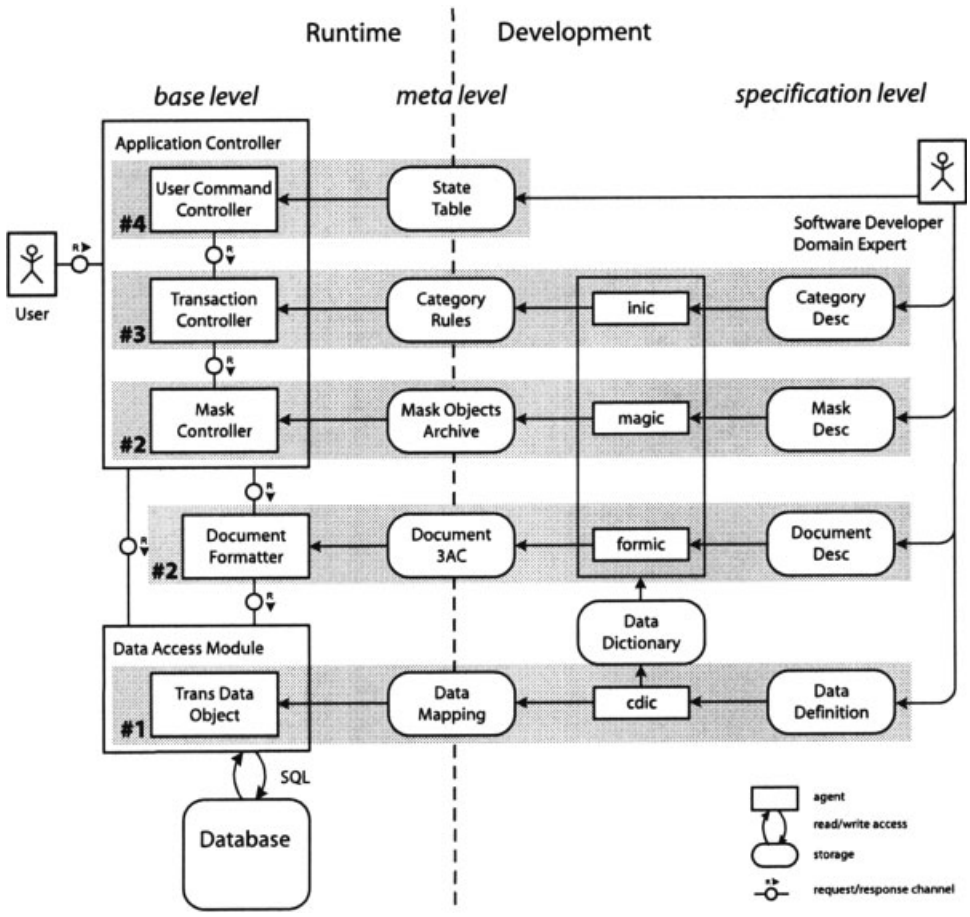

Figure 1. Overview of the architecture (FMC notation)

development on the right. The behavior of the system is described on the *specification level*. These domain descriptions are transformed into the *meta-level* objects that control the components of the *base level* at runtime.

The `Application Controller` starts the application and passes control to the `User Command Controller`. The `User Command Controller` reacts to user commands invoked by the application menu according to the `State Table` that describes the dynamic behavior of the application. When the user chooses a category and opens a civil transaction for processing, the `Transaction Controller` reads from the `Category Rules` how to handle this civil transaction.

The `Transaction Controller` passes control to the `Mask Controller` when the user opens a mask to enter data. The `Mask Controller` dynamically generates presentation objects from the `Mask Objects Archives` and shows them on the screen. It uses data references to establish a connection between the input fields and the `Trans Data Object`, containing the data for the current civil transaction. (The same mechanism is used for dialogs, but neither is shown in the figure nor discussed in this paper.)

When the user issues the command to print documents, the `Application Controller` passes control to the `Document Formatter`.

The `Mask Controller` and the `Document Formatter` need the data of the current civil transaction. Whenever data access is necessary, they demand this service from the `Data Access Module` or, more specifically, the `Trans Data Object`.

During development, domain experts specify Mask Descriptions and Document Descriptions in domain-specific languages. The compilers `magic` and `formic` translate these descriptions into `Mask Objects Archives` and `Document Three-Address Codes` respectively.

The rules that tell the `Transaction Controller`, which masks and documents are required according to the constellation of the current civil transaction are specified in `Category Descriptions`. The preprocessor `inic` translates them into `Category Rules`.

Software developers specify the `Data Definition` of the database. The compiler `cdic` translates it into a `Data Dictionary` and a `Data Mapping`. Using the `Data Dictionary`, `magic`, `formic`, and `inic` verify the data references in the descriptions. The `Data Mapping`, read by the `Trans Data Object`, provides information on tables, primary keys, and integrity constraints.

3.3. Remarks

The meta-level architectural pattern is discussed e.g. by Buschmann *et al*. (1996) and Yoder and Johnson (2002). These discussions focus on meta-object models that determine the creation and behavior of base-level objects at runtime by a reflection mechanism. In our mechanisms, meta-data written in domain-specific languages describe meta objects. Reflection mechanisms have their place in the implementation of the controllers at the base level.

Both the domain knowledge *and* its transformation into workflows should not be hardwired into the code. We tried to find the most convenient way. The domain experts describe all layers of the application on the specification level – the data, the input and output, and the control – and these descriptions are transformed to control the base level. This solution is generic by means of mask and document generators. Their application, however, is very specific to the domain.

Interactive systems are often described by a *layer* architecture. Basically, three conceptual layers are distinguished: the presentation, the domain logic, and the persistence layer (e.g. as described by Fowler (2003)).

Our architecture has layers in two dimensions: with respect to base and meta-levels as well as concerning responsibilities. In both the base level and the meta-level, we distinguish presentation, domain logic, and persistence. The most interesting aspects of the meta-level architecture with respect to layering are as follows:

- The layers in the meta-level are tightly coupled, supporting the required domain-specific dependencies. The construction of the meta-level follows the way domain experts describe the processing of civil transactions. For this reason, we avoid indirection wherever possible.
- The layers in the base level are extremely decoupled. Engines in each base-level layer interpret the meta-level descriptions at runtime. These engines do not hardwire domain-specific

logic, but process the meta-level descriptions. The architecture provides access points to the layers; the engines do not depend on each other in any other way. Consequently, the base-level engines can operate in different combinations in the various products of the product line.

4. MECHANISMS

4.1. Mechanism #1: Data Reference and Access by Symbolic Names

References to data items are needed in the meta-level to specify data input, document contents, and workflow. We want to reference and access data items by names. We want to ignore where data is stored and how.

4.1.1. Solution

Within a civil transaction, a symbolic name uniquely identifies a data item. The symbolic name determines the access path to the data in the database. At runtime, a dynamic data object serves as a container for this data and is responsible for its persistence.

Data Model. To define a naming convention for data items, we need a convenient data model. In our application, each civil transaction is assigned a unique *root entity* whose primary key is called `Civil Transaction Identifier (vid)`. We can then organize the data involved in the civil transaction in entities so that each entity has a 1:1 relationship or a cascading 1:n relationship to the root entity. Each entity type has a *compound primary key* whose first item references the `vid`.

Figure 2 shows the principle in an example: Root table `hbrt` contains one row for the marriage identified by `vid 1234`. In table `hbvp`, there is one row for fiancé Schneider and one row for fiancé Bergmann, distinguished by `ptype e` and `s` respectively. `anzve` holds the number of previous marriages. Mrs. Bergmann, now divorced, was previously married to Mr. Olthoff, while Mr. Schneider is unmarried. So table `hbve` has one entry. This row is linked to `hbrt` by the `vid` and to `hbvp` by the `vid` and ptype `s`.[2]

[2] Please do not be troubled by cryptic abbreviations such as hbvp. They are reasonable abbreviations for Germans. Our domain experts love them.

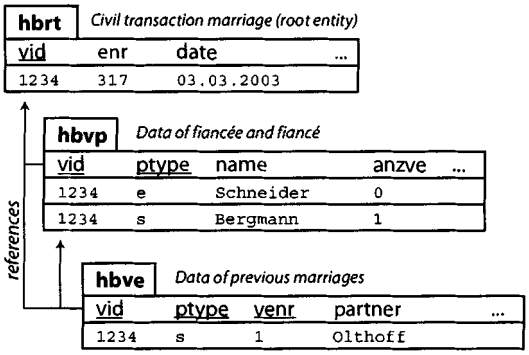

Figure 2. A part of the data model illustrating the principle of relationships

Data Reference. A *symbolic name* consists of the *row identifier* (i.e. the name of the entity type and the key information) and the *field identifier* (i.e. the name of the attribute). A dot separates the row and field identifiers. Using the example of Figure 2, the symbolic name

```
hbve[s][1].partner
```

references `Olthoff`, the name of the previous spouse of Mrs. Bergmann. It corresponds conceptually to the SQL statement

```
select partner from hbve where vid=? and ptype='s' and venr=1.
```

Data Access. The data of one civil transaction is encapsulated in a `Trans Data Object (tdo)`. Assisted by the `Data Mapping`, `tdo` translates the symbolic name into a data access path. To continue with our example, `tdo` reads in the `Data Mapping` that the compound primary key of `hbve` is composed of the attributes `vid`, `ptype`, and `venr`, as indicated in Figure 2. The `vid` is the identifying attribute of the `tdo`. At runtime, its value replaces the placeholder `?`. So, `tdo` has all the necessary key information to access the data of `hbve[s][1].partner`.

The `tdo` holds the data of a transaction in memory. We implemented two strategies to load the data from the database into the `tdo`:

- Load all data rows identified by the same `vid`. This gives a full in-memory copy of the data belonging to this civil transaction.
- Load only those data rows requested by the application (lazy load). At each request, `tdo` checks whether the requested data is already in memory (i.e. contained in the current `tdo`).

Which strategy we choose depends on the infrastructure. If the database connection is fast, we can load all data rows when opening the civil transaction. If, however, the database connection is rather slow, we will use lazy load. Otherwise, the user would have to wait too long for the first mask. It is even possible to mix both strategies, that is, load the core data on creation of the `tdo` and load the remaining data as requested by the application.

Concurrency Control. The data values in the `Trans Data Object` (i.e., in memory) have to be synchronized to the database. The `Trans Data Object` intercepts all database access and keeps a log of the state of the data in memory. Therefore, the `Trans Data Object` has to perform long-duration transactions (Silberschatz *et al*. 2002).

As databases usually do not support long transactions, we implemented a check-out/check-in mechanism used by the `Data Access Module`. On creation of a `tdo`, the current user is recorded in the corresponding root entity, and all other users are denied access.[3]

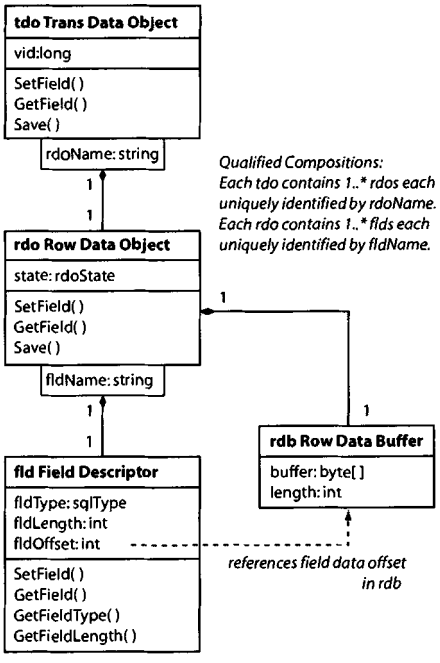

Figure 3. Code structure of the `Trans Data Object` (UML notation)

[3] This is in no way a restriction. It meets the working procedure in the offices.

4.1.2. Implementation

Figure 3 shows the structure of the implementation of `tdo`: The class `Trans Data Object (tdo)` is a container of `Row Data Objects` (`rdos`). Each object of class `tdo` is uniquely identified by the `vid` given on construction of the `tdo`. `tdo`'s methods to retrieve and store the values in the database delegate this responsibility to the `rdos`, `tdo`'s components.

Each object of class `Row Data Object` stores one row of data. It has an attribute `rdoName` whose value is the row identifier. An `rdoName` is unique within the `rdos` contained in one `tdo`.

Each `rdo` consists of an `Row Data Buffer` and a list of `Field Descriptors`. The `Row Data Buffer` stores the data. It is allocated dynamically at runtime and structured by the `Field Descriptors`. Each `Field Descriptor` contains the meta-information of the corresponding database field: name, type, and length. This meta-information is retrieved from the system catalog of the database at runtime. The `Field Descriptor` is used to make all necessary type conversions and to inform the application about the type of a data item in the database.

The `rdo` keeps track of its *state* with respect to the database in its attribute `rdoState`. The state of an `rdo` is characterized by the statechart in Figure 4.

When created, the `rdo` is initialized from the database. If the corresponding row exists in the database, the data are loaded, and the state of the object is PERSISTENT. Otherwise, it is TRANSIENT.

The methods of the `rdo` manipulate the data in the `Row Data Buffer` and change the state of the `rdo` accordingly. The methods SetField()

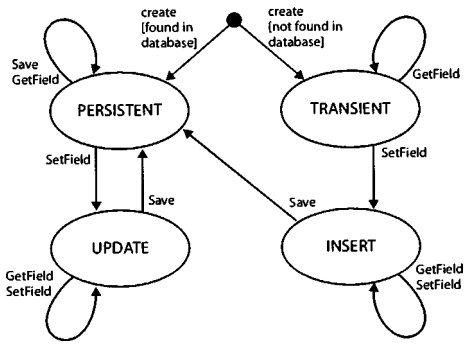

Figure 4. Data persistency and synchronization (UML statechart notation)

and Save() change the state of the rdo to ensure the correspondence to the state of the data in the database.

The method Save() chooses the appropriate action on the database according to the state of the rdo. In state UPDATE Save() performs an update of the data row. In state INSERT it inserts a new row in the table. The primary key of the new row is given by the rdoName. In state PERSISTENT or TRANSIENT, nothing needs to be done.

4.1.3. Discussion

The data model and data reference are *simple:* The data model consists of 1:1 and 1:n relationships only. It is mapped directly to the data definition of relational databases. Software developers and domain experts can easily keep both the data model and data reference in mind.

The data reference is *database-system independent* and *programming-language independent:* The symbolic name is a character string. It introduces an indirection between the data reference and data access. The data reference assigns a meaning to a database field – it establishes the necessary coupling in the meta-level. The symbolic name may be translated to any database-access technique.

Changes in the Data Definition affect the users of the data (Mask Descriptions and Document Descriptions), while the data-access mechanism remains unchanged.

Data access is *generic and dynamic*: tdo provides a data-access mechanism and a data container, but does not know which data to access. This information is contained only in the meta-level: the symbolic names. tdo resolves symbolic names at runtime.

Data access is *portable*: We had few problems with incompatibilities of SQL databases because we only used a small subset of SQL. We were even able to implement this SQL subset on a navigating database. A simple data model has led to database-system independence.

The basic idea of the Trans Data Object can be extended easily to a check-out/check-in mechanism with a more sophisticated data mapping and naming convention. An option is to use an XML data model and a subset of XPath or XQuery as a naming convention to identify individual data items or even groups of them.

4.1.4. Related Mechanisms

Mechanism #1 is the basis for the other three mechanisms. They use the symbolic names.

4.1.5. Remarks

The concept of the Trans Data Object and the use of a statechart to control the state of the rdos was inspired by Rumbaugh *et al*. (1991, Chap. 20).

The patterns in Fowler (2003, Chap. 11 and 13 are closely related to our approach: the Unit of Work, Metadata Mapping, and Repository patterns share concepts with the Trans Data Object. However, we use the Trans Data Object to store and retrieve data values whose access paths are given *dynamically* – by symbolic names, in fact. So the Trans Data Object is more restricted in terms of the data model, but within that more generic.

The disconnected data set of Microsoft's ADO.NET (see e.g. Thai and Lam (2001)) has many similarities to the Trans Data Object. We already mentioned that an XPath-based naming convention could be used instead of ours.

4.2. Mechanism #2: Input and Output Control by Domain-specific Descriptions

The data necessary to process a civil transaction is collected in a series of masks, i.e. input forms. The Mask Controller reacts on input immediately and guides the user through the transaction, depending on the specific situation.

The data is then used to print all required documents. Usually, a civil transaction results in 5–20 documents, each comprising 1–4 pages.

4.2.1. Solution

> *For each transaction, platform-independent mask and document descriptions define which data is entered and how forms are filled out. They are written in domain-specific languages. At runtime, the compiled descriptions control the layout and behavior of masks and the content and formatting of documents.*

Mask and document descriptions are written in languages that we designed according to the needs of the domain. Certain *functions* of the languages provide the domain-specific features, notably

- properties of fields on masks and documents

- processing of input
- formatting of output.

We continue describing the solution and implementation for masks and documents separately.

4.2.2. Masks

Concept. An `Mask Description` defines a static structure and dynamic behavior:

- The layout of the mask is defined by placing labels and input fields according to a column raster.[4]
- Each input field is assigned the symbolic name of the data item that stores the input.
- Input is assisted in certain fields by domain-specific functions, such as date functions, auto-completion, and calculations.
- Actions are triggered by user interaction events (enter or quit the mask, enter or quit a field, even type certain characters in a field). Actions are domain specific and may depend on parameters, data in input fields, or data of the current transaction:
  - (pre-)fill fields with data
  - disable/enable fields
  - check the logical state of input.

Implementation. The compiler `magic` translates the platform-independent `Mask Descriptions` into the platform-dependent `Mask Objects Archives`. For example, for MS Windows, these are serialized C++ objects. At runtime, the `Mask Controller` uses the `Mask Objects Archives` to control the mask.

The `Mask Controller` shown in Figure 5 is designed according to the Presentation Abstraction Control (PAC) architecture (Buschmann *et al*. 1996). The Presentation manages the screen, intercepts arriving messages, and routes them to the Control. The Abstraction maintains the data via the `Trans Data Object`. Moreover, the Abstraction keeps all meta-level information to provide the intelligence needed to handle user interactions that require executing functions, disabling fields, and so on.

[4] A template defining the layout of the masks was developed by a UI designer.

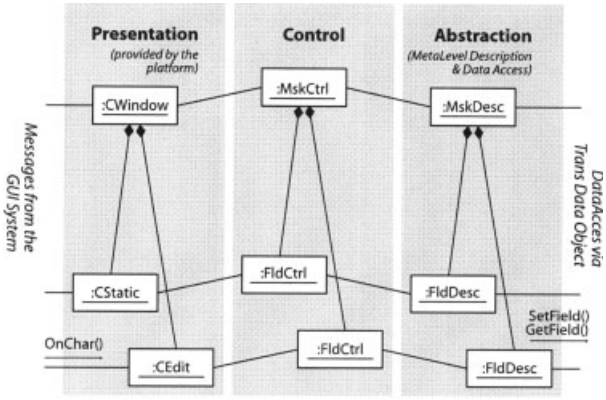

Figure 5. The PAC architecture of the `Mask Controller`, exemplified with MFC (UML notation)

The Control – the mediator between Presentation and Abstraction – recognizes the input, asks the Abstraction what to do, executes the appropriate functions, passes the result back to the Presentation, and returns control to the operating system. With this interaction between the components of the PAC architecture, the behavior of fields and masks is not hardwired into the code, but determined by the meta-level information contained in the Abstraction.

4.2.3. Documents

Concept. A `Document Description` defines layout and content:

- A set of fields defines the layout of the document. Each field is defined by a composition of rectangles, namely, their positions on the document, length, and height.
- A set of formatting rules defines the properties of each field. A field may have several of about 20 properties, notably
  - generic properties (e.g. flush left/right)
  - domain-specific properties (e.g. the number of in-between lines).
- Assignments define the content of each field.
  - A field may contain static text, data items, or any composition of these.
  - The filling of fields may depend on parameters or on any data of the current transaction. The language provides if-then-else and switch-case for that.

For all this, the language provides about 60 functions, implemented in the host language. Figure 6 gives an idea of the language.

```
if (hbeh.ehename != "") then
     f1 = hbvp[$1].name + " heiratet am " + kdat(hbrt.date)
endif
```

Data reference with Parameter
Assignment
Form field reference
Constant
Data reference
Function
String concatenation

Figure 6. An example from a Document Description

Implementation. The compiler formic translates a Document Description into platform-independent Document Three-Address Code (see Figure 1). One instruction consists of a destination, two operands, and an operator.

At runtime, the Document Formatter composes a document from the Document Three-Address Code and the data provided by the Trans Data Object (Figure 7). First, the Document Formatter loads the Three-Address Code and the Symbol Table into memory. With the help of the tdo, it replaces all data references in the Symbol Table with the current data values. Then, the Virtual Machine processes the Three-Address Code. Using functions in the Function Pool it computes the content of each field and stores it in its destination in the Symbol Table. The functions are implemented in the base level, comprising not only generic functionality but also domain-specific features. After processing of the Three-Address Code, the Symbol Table contains all the information necessary to produce the document (i.e. field contents and properties). Finally, the Document Formatter traverses the Symbol Table and produces a virtual document in memory. This document contains the completely processed content in a device-independent format tagged with printing instructions. When printing the document, these instructions are replaced with the commands for the current printer.

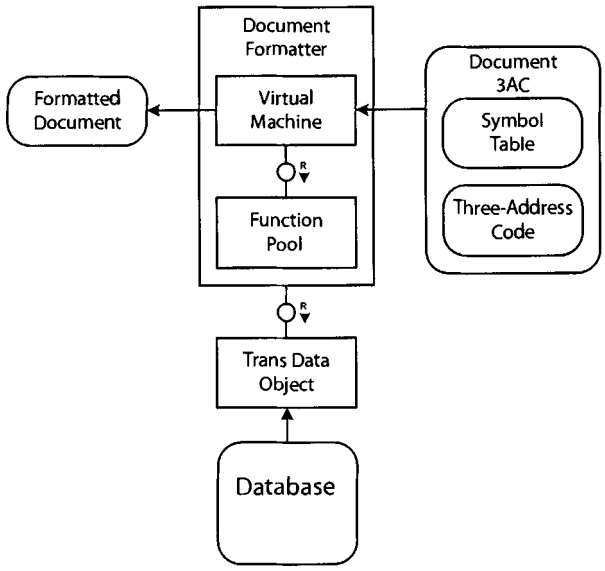

Figure 7. Compositional structure of the Document Formatter

4.2.4. Discussion

In each Mask Description and Document Description, highly valuable domain expertise is captured and is readable, maintainable, and available for domain experts and software developers. Each description serves as a specification, code, and documentation. It is also guaranteed to be up-to-date, because this domain knowledge is captured in no other place.

Mask Descriptions and Document Descriptions inherently expose a high degree of correctness because the languages guarantee consistency and prevent programming errors. Testing is reduced to a black-box test of functionality. (See the discussion of quality attributes in Section 5.5)

'Coding' masks and documents comes close to specifying because the languages are declarative. They are procedural only where necessary. Given these tools, our domain experts can easily translate concepts they have in mind into the descriptions. After a while, they got used to think in the functions of masks and documents.

4.2.5. Related Mechanisms

Mechanism #1. Masks and documents reference and access data via the Trans Data Object (tdo). Their data references must correspond to database fields. The correctness of the data references in the Mask Descriptions and Document Descriptions is checked at compile time: formic and magic look up all symbolic names in the Data Dictionary and will report an error if a symbolic name is not there. This compile time check prevents failures during the late binding of the meta-level objects at runtime.

Masks and documents can be parameterized. For example, the same Mask Description can collect similar data for different people if a key attribute, such as ptype, is a parameter. Likewise,

the same `Document Description` is used for different people or different recipients.

Mechanism #3. Depending on the specific situation, the `Transaction Controller` determines which masks to present for input and which documents to offer as output.

Mechanism #4. The `Mask Controller` is part of a Chain of Responsibility (Gamma *et al*. 1995): Messages resulting from user interaction are routed to the Control component of the `Mask Controller`. If the required reaction involves only the current input field, it is handled by the Field Controller (FldCtrl). If several fields are involved, the `Mask Controller` (MskCtrl) is responsible; if another mask is involved, the `Transaction Controller` comes into play. Finally, the `User Command Controller` is the top level of this chain.

4.2.6. Remarks

We use the Presentation Abstraction Control architectural pattern presented in Buschmann *et al*. (1996) for the design and implementation of the user interface control components. The dependencies on the platform are encapsulated in the Presentation component. Abstraction and Control are platform independent.

The concepts used for the design of the `Document Three-Address Code` and the implementation of `formic` come from the 'classical' sources (Aho *et al*. 1986, Holub 1990).

Both compilers, `magic` and `formic`, are made with lex & yacc. Subroutines in formic are implemented with the m4 macro processor.

4.3. Mechanism #3: Application and Domain Logic by a Rule Engine

The handling of a civil transaction is organized as a workflow: it leads the user through a series of masks. It results – after a juridical check – in printing several documents. The order of the masks and the selection of the documents depends on the specific situation (i.e. on personal data).

The structure of the workflow is common to all categories of civil transactions, but the concrete series of masks and documents within a category is domain knowledge. As such, it would be inappropriate to program the workflow in technical code. Instead, we continue to keep the domain knowledge in the meta-level.

4.3.1. Solution

> *The workflow is controlled by properties. The properties describe the series of masks and documents and the conditions on which masks should be presented and documents should be printed. These domain-specific properties are described in the meta-level. At runtime, a rule engine controls the workflow using these descriptions.*

Each mask or document has the following properties: a name, parameters that are substituted on invocation, and conditions that restrict its use, depending on the current data of a civil transaction.

The complete workflow of a category of civil transactions consists of a list of all possible masks and documents and the conditions.[5]

The conditions – logical expressions – are the heart of the application and domain logic. All constraints are grouped in classes of conditions. Each class comprises all permissible states of interdependent data items. This is called a *logical class* and the *logical state* within a class. Each condition can be referenced by its logical class and logical state.

For example, in processing a marriage, the logical class 'marital status' permits among others the logical states 'single' or 'divorced, number of previous marriages > 0', whereas 'married' (obviously) is not allowed. The logical expressions for the fiancée in a `Category Description` are

```
hbvp[s].famstand=="ledig"
     && hbvp[s].anzve==0
hbvp[s].famstand=="geschieden"
   && hbvp[s].anzve>0[6]
```

4.3.2. Implementation

A `Category Description` contains the properties of a category (i.e. the series of masks and documents and the logical conditions). `Category Descriptions` are files in the ASCII format, structured by *sections* and *tags*.[7]

[5] Some masks or documents are *repeated* several times. The number of repetitions depends on the current data (e.g. the number of children). Our descriptions allow for specifying repetition, but we omit the details in this article.

[6] famstand = marital status, ledig = single, anzve = number of previous marriages, geschieden = divorced

[7] The style of Windows ini-files. Our product line started long before XML was invented.

First, `inic` checks the syntax of data references against the `Data Dictionary` (see Figure 1) to guarantee the correctness of all data references. Then, `inic` transforms the conditions into Reverse-Polish Notation. The preprocessor `inic` translates a `Category Description` into a `Category Rules` file. This simplifies and speeds up the evaluation of the expressions at runtime.

At runtime, the `Transaction Controller` starts the processing of a civil transaction by transforming the `Category Rules` into runtime objects. During input, the rule engine evaluates the conditions to control the behavior of the workflow.

4.3.3. Discussion

Using the concept of logical classes and states, we initially intended to reach a more declarative way of describing the domain logic than we finally did. International civil law brings up a complexity that is hard to capture in a comprehensive set of logical expressions. Domain experts often prefer to design the processing of the civil transaction in terms of controlling the input on the masks. Many regulations of the domain are formulated in this manner (Schmitz and Bornhofen 2001).

4.3.4. Related Mechanisms

Mechanism #1. `Category Descriptions` reference data items by their symbolic name. The rule engine uses the `tdo` to resolve these references and access the values of the data items.

Mechanism #2. The `Transaction Controller` uses the rule engine to determine the series of masks and documents and gives control to the `Mask Controller` and the `Document Formatter`. In particular, it provides the parameters and the reference to the `tdo` for the processing of masks and documents.

The `Mask Descriptions` refer to the `Category Rules`, which describe the integrity conditions of input values. At runtime, the `Mask Controller` evaluates the conditions by means of the rule engine.

4.4. Mechanism #4: User Command Control by a Finite State Machine

So far, we have described how data is entered on masks and is used to print documents. The controlling of the application is still missing. For example, before users can work at a civil transaction, they choose the category from the menu and either open an existing transaction or set up a new one. After entering the data, they issue a printing command.

4.4.1. Solution

> *Describe top-level user commands as events that trigger actions and move the application from one state to another. At runtime, a finite state machine reacts to the events by executing the actions and transitions.*

We describe the behavior of the application by a statechart. It consists of sets of states, events, actions, return codes, and two types of transitions. It is a variant of the statecharts introduced by Harel (1987).

In a *state*, an *event* may occur. State plus event define the *action* to be executed. Each action issues a *return code*. An action plus a return code define the follow-up state. Figure 8 gives an example. Both in CATEGORY STATE and in CIVIL TRANS STATE, the user may `open` a civil transaction for processing. If the user confirms the `Open Civil Transaction` dialog with OKAY, he will arrive in CIVIL TRANS STATE. Otherwise, he will return to the state where he came from, indicated by an `H`.

4.4.2. Implementation

The triggering of a menu command is routed to the `User Command Controller` whose finite state machine runs the statechart. The statechart is defined by a *state table*, a static data structure that contains the flattened statechart, i.e. nested states are transformed into unnested ones.

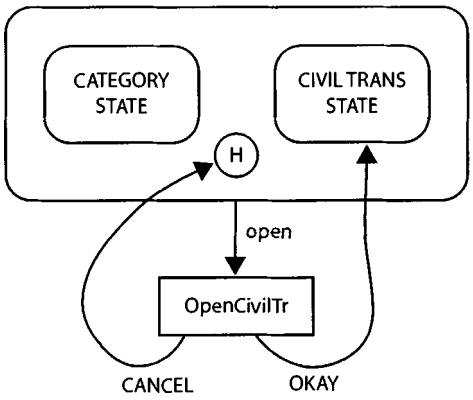

Figure 8. Part of the user command control statechart

At runtime, the `User Command Controller` uses this `State Table` to control the behavior of the application. Events come from user interactions. Actions are methods of the `Application Controller`.

4.4.3. Discussion

Ergonomics. Modeling the control flow of a program with a statechart supports software ergonomy. The statechart maps the working procedure in the office to the control flow of the program. The result is a clear, transparent behavior of the program that users understand effortlessly. Our users have never asked questions such as 'where am I?' or 'how did I get here?'

It was possible to keep the statechart simple because all categories and transactions are processed according to the same pattern (see mechanisms #2 and #3). All of them are handled with the same actions.

Static data structure. We decided to implement the `State Table` in a static data structure instead of a dynamic one because statecharts rarely change in our products.

Product Line. The description of the statechart in the meta-level decouples application control from GUI infrastructure. For example, on the MS Windows platform, we redirect MFC's message map to the `User Command Controller`'s `State Table`.

The implementation of the finite state machine is reusable with other statecharts. For example, the functionality of the software can be reduced by removing states and transitions from the statechart. Variant products can sometimes be obtained by providing a different `State Table`.

4.4.4. Related Mechanisms

The `User Command Controller` is not directly connected to mechanisms #1, #2, or #3. The repository `tdo` facilitates the use of the finite state machine: actions share data in the repository.

4.4.5. Remarks

There are several techniques for implementing statecharts, e.g. in Samek (2002) and Horrocks (1999).

As shown in Figure 8, our statechart is a bipartite graph, where states are distinguished from actions. This notation visualizes behavior clearly and coherently. We used it long before UML came up. Nevertheless, this notation can easily be made UML conformant by using stereotypes to distinguish states from actions.

4.5. Applicability of these Mechanisms

Our architecture and mechanisms are applicable for software systems that process interactive transactions and follow certain patterns. Examples of such systems are

- tax return
- insurance claims
- loan applications
- negotiating contracts.

Generally, our mechanisms will suit if

- transactions have to be processed and follow certain patterns
- data can be modeled according to our principles
- many different masks and forms are needed
- several products are developed
- domain logic should survive when technology changes and vice versa.

The basic idea – let domain-specific descriptions control the application – can be used to design many applications. We imagine process control systems, workflow management systems, automotive or avionic control systems. In embedded systems, the interpretation of descriptions may be too slow and the readable descriptions may consume too much memory. Here, the descriptions can be compiled to save runtime resources.

5. META-LEVEL ARCHITECTURE AND PRODUCT LINES

A product line architecture exploits commonality, supports the management of variability, and eases traceability. Commonality is best exploited by a large-scale reuse of core assets. Variability is realized more easily if it does not require modification of code or management of complex configurations. The shorter the trace, the easier it is to handle the traceability from requirements to implementation and documentation.

As an example, we discuss the contribution of our product-line architecture, the domain-specific languages, and the four mechanisms

- #1: Data Reference and Access by Symbolic Names
- #2: Input and Output Control by Domain-Specific Descriptions
- #3: Application and Domain Logic by a Rule Engine
- #4: User Command Control by a Finite State Machine

to the realization of commonality, variability, traceability, and product derivation.

Designing a software architecture is about making decisions, and these decisions are made on the basis of the quality attributes that the system shall expose. Every architect is put in charge of predicting the qualities, which his architecture will introduce. It is therefore interesting to study the influence of a meta-level architecture and domain-specific languages on quality attributes. We do so in Subsection 5.5.

5.1. Commonality

In our product line, each product's architecture is identical to the meta-level architecture shown in Figure 1. There are no derived architectures. Every component and relation shown in the architecture is part of every product. The distribution of responsibilities between the base level and the meta-level is always the same, and the four mechanisms are used in all products.

- Mechanism #1: Each product needs a database, data reference, and data access. Nevertheless, data reference by symbolic names is database independent.
- Mechanism #2: Each product comprises input of data and output of documents.
- Mechanism #3: In each product, the `Transaction Controller` controls the workflow and `Category Descriptions` are compiled into `Category Rules`.
- Mechanism #4: Each product needs to react to user commands (e.g. menu commands or shortcuts). The `User Command Controller` receives and processes user commands according to the `State Table`.

The domain-specific languages for mask and document descriptions and the symbolic names are appropriate for the domain.

5.2. Variability

We distinguish between variability of the domain and variability of technology. Our architecture distinguishes clearly where they take place.

5.2.1. Realizing Variability of the Domain

Variability of the domain is primarily realized at the specification level, see Figure 9.

Mechanism #1. Each product needs a `Data Definition`. This `Data Definition` is kept in one file that belongs to this product. If products share the same `Data Definition`, they will share this file.

Mechanism #2. `Mask Descriptions` and `Document Descriptions` are product specific. That is, to each product belongs a set of `Mask Descriptions` and a set of `Document Descriptions`, and each description is in one file. Some descriptions can be used for more than one product. This is handled by the configuration management system, where two products can share the same configuration item.

The description languages need extension for some products (e.g. Austrian registry offices need extra formatting which is not used in Germany).

`Mask Controller` and `Document Formatter` will need adaptation only if a certain function needs different implementations for a product.

Mechanism #3. `Category Descriptions` are product specific, that is, to each of our products

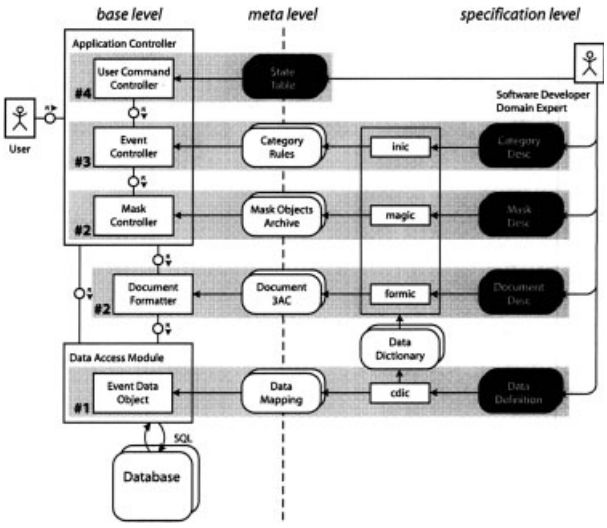

Figure 9. Realizing variability of the domain

belongs a set of files containing the `Category Descriptions`.

Mechanism #4. The `State Table` is product specific. We decided to describe the `State Table` as an array in a header file because our state tables change rarely. If they changed regularly, we would prefer to put their description in the specification level.

5.2.2. Realizing Variability of Technology

Figure 10 shows where variability of technology happens.

Mechanism #1. Different database technologies require different implementations of the `Data Access Module` and the `cdic` tool (e.g. data access on a navigating database differs from data access on an SQL database.) The `Trans Data Object`, however, which resolves the data references, is used without adaptation in all products.

Mechanism #2. The `Mask Controller` depends on a GUI infrastructure to a high degree, and so does `magic`. `magic` generates the `Mask Objects Archives` appropriately. The mask description language depends to a certain degree on the possibilities the GUI provides. We kept both `magic` and the mask description language downward compatible.

Printing depends on printing technology, but the `Document Descriptions` and the `Document Formatter` do not.

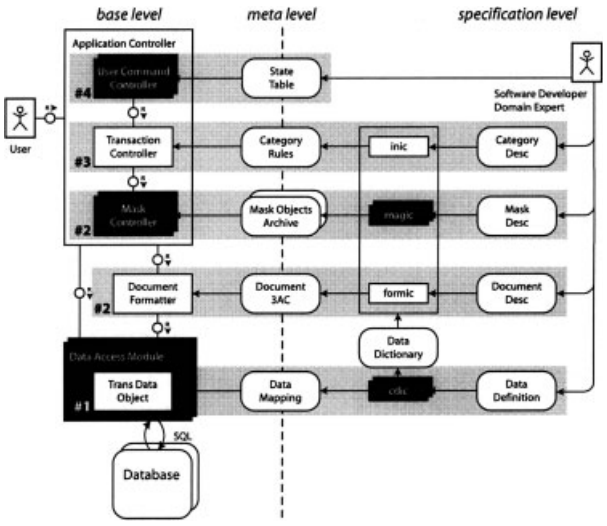

Figure 10. Realizing variability of technology

Mechanism #3. This mechanism is independent of technology.

Mechanism #4. The `User Command Controller` depends on the GUI. Events resulting from user interactions have to be redirected to the `User Command Controller`.

5.3. Traceability

In our product line, requirements come from law and from customers. Requirements concerning technology, such as 'we want to use database X' or 'we want to use network Y', are easy to handle.

Requirements concerning the domain are spread across thousands of details and are sometimes difficult to understand. If these requirements are realized in one of our products, they will be specified in `Mask Descriptions`, `Document Descriptions`, and `Category Descriptions`. Comments on their origin can be added. The descriptions serve as specification, code, and documentation. They control what the software does and they answer all questions about why, when, and how the software does something. Experienced domain experts can read them, write them, understand them, and use them as the single source of truth. Traceability is therefore trivial.

5.4. Derivation and Configuration of Products

We describe the derivation of products in four examples:

- **Variants due to German state law.** All variants for German registry offices use the same `Data Definition`. `Mask Descriptions`, `Document Descriptions`, and `Category Descriptions` differ. Differences are realized either by various descriptions or by conditional expressions in a description. During installation, the software is customized.
- **Variant due to Austrian law.** The `Data Definition` is different from Germany, and so are `Mask Descriptions`, `Document Descriptions`, `Category Descriptions`, and the `State Table`. All differences are located in the specification level, as shown in Figure 9. Disjoint sets of descriptions are deployed for Austria and Germany.
- **Variants due to database technology.** The products in our product line run on different

database systems. The application programming interfaces of the databases determine if the base level components can be reused. SQL databases support a call level interface more or less according to the SQL standard. We implemented one variant of the `Data Access Module` for all of them. Customization during installation handles the subtle nuances. Non-SQL databases (that we used for cost reasons) required a variant of the `Data Access Module`. The appropriate `Data Access Module` for a product is chosen at link time. In general, the reusability of base level components depends on the adaptability of the underlying technologies.

- **Variants due to operating system and user interface.** Since the first generation of our product line, the evolution of operating systems and user interfaces has taken two major steps: from an alphanumerical, semigraphical UI under DOS/Unix to the graphical user interface of Windows, and then to web-based interaction forms. Both steps required the reimplementation of components of the base level (see Figure 10). Our software products are adapted to different technologies by exchanging base level components, which leaves the domain functionality untouched. Variants of base level components are chosen at link time.

5.5. Impact on Quality Attributes

We discuss the impact of a meta-level architecture and of domain-specific languages on the quality of a software product or a software product line. We follow the classification of quality attributes in Bass *et al*. (2003). In each case, we give an example from our product line.

5.5.1. Modifiability

Modifiability can be achieved by localizing expected changes and preventing the ripple effect. In a meta-level architecture, the separation of concerns between the meta-level and the base level allows the modification of artifacts independently at different levels, in particular,

- to make modifications at the specification level.
- to change the base level, while leaving the specification level untouched, and vice versa. Making changes at the base level takes more effort, but is supposed to happen less frequently.

To ease these kinds of changes was one of our major architectural goals. The four mechanisms are designed to support the modifiability of our products.

On the other hand, the price we pay for the separation of concerns between the meta-level and the base level is an architectural commitment.

Modifiability can also be improved by simplifying the deployment process, in particular, by deferring the binding time. A meta-level architecture provides full flexibility for binding times.

Our mechanisms #2 and #3 decouple changes of domain-specific descriptions from the construction of the base-level components. Meta-level objects are replaceable at runtime.

Domain-specific languages defined by a grammar, as the languages in our mechanism #2, expose a high degree of modifiability. The syntax can be extended, and additional features can be introduced in a downward compatible manner. The language can be used in different environments.

On the other side, the conceptual scope of a domain-specific language is hard to modify. All the more, the language must capture the essence of the domain requirements. We discuss this point in Section 6.

5.5.2. Performance

A meta-level architecture introduces an indirection, which slows down performance. Symbolic information is evaluated at runtime via a reflection mechanism. Smart implementations can, however, diminish these effects. For example, the implementation of the `Document Formatter` has very little overhead because the `Document Three-Address Code` generated by the compiler is optimized for performance.

It is not possible to make a general statement about the impact of domain-specific descriptions on the performance of a system. Generated code should be faster than interpreting the descriptions, as we do in mechanism #2.

In our architecture, the resolution of the symbolic names by the `Trans Data Object` is the critical point with respect to response time. In fact, there is a trade-off between performance and modifiability. We decided for adaptability. In an interactive system like ours, the environment (e.g. networks or databases) often has more impact on performance than the architecture, regardless of the decision one makes with respect to meta-level mechanisms.

5.5.3. Testability

The testability of a software system is influenced by both a meta-level architecture and domain-specific languages.

A meta-level architecture allows us to decouple the testing of the base level from the testing of the meta-level. The built-in separation of concerns sets a clear dividing line for testing. The following questions can be answered independently:

- Do the compilers generate correct meta-level objects according to their specification?
- Do the base-level components work correctly under the control of the meta-level objects?
- Are the domain descriptions correct?

The architecture also facilitates the locating of defects. Given a certain failure, it is easy to say in which component the defect is. Just as the levels decouple technology from domain logic, they decouple responsibilities of software developers from those of domain experts.

Domain descriptions written in domain-specific languages defined by a grammar give rise to straightforward test strategies. The primitive elements of the language can be checked independently and the ways to combine these primitive elements are well defined. If the implementation of the language prevents side effects, testing is further facilitated. For example, in our languages it is impossible to generate recursive invocation.

5.5.4. Availability

A meta-level architecture or domain-specific languages do not directly affect the availability of the system. There is, however, an indirect effect because such a meta-level architecture allows us to design and implement more testable and less faulty software.

In our development process, we classified faults in the base level in

1. defects resulting from coding errors
2. effects resulting from unexpected user interaction
3. failures arising from external effects, such as hardware errors, or shortage of resources.

Meta-level architectures and domain-specific languages can prevent faults of type 1 and guard against type 2 and 3:

- Meta-level software product line architectures lead to frequent use of the base level components. Hence, these components achieve a high level of stability. In fact, our base level was defect-free.
- By explicitly separating base level and meta-level, we achieved excellent testability and easier detection of defects.

5.5.5. Security

A meta-level architecture opens additional security holes, in particular, if interpreters are used in the base level.

SQL injection is a well-known example of the false attribution of meta-level objects to infiltrate a system (Whittaker and Thompson 2004, Page 48). As a consequence, the implementation of a meta-level architecture requires special attention to security. For example, digital signatures can assure the authenticity of meta-level objects.

Owing to the organizational environment in which our products are used, we did not need to take any special precautions.

5.5.6. Usability

A meta-level architecture leads to a *uniform behavior* of the system. Both developers and users understand it easily. The software offers the user a clear and consistent mental model. It is in the nature of this architecture that features of the same type behave in the same way. They appear in the meta-level many times, but their processing is implemented only once.

On the other hand, sometimes users want variants that may be hard to realize. Special cases, exceptions, or ad hoc variants are impossible or possible only with more effort. Careful positioning of the distinction line between base and meta-level will allow the implementation of the variants needed by the specifications.

Therefore, the design of the domain-specific languages is crucial for usability and adaptability. Only if the languages used in the specification level of a meta-level architecture capture the essence of the domain will the software give the user satisfaction, because only then will the user perceive the system as a solution for his tasks. Accordingly, the main question we must answer if we want to achieve a high level of usability is how to find the suitable language that meets the specific properties of the domain. This is discussed in the next section.

6. ON THE DESIGN OF DOMAIN-SPECIFIC LANGUAGES

6.1. Finding Patterns in the Domain

The key for designing domain-specific languages is to find *patterns in the domain* that allow us to describe an intended behavior of a software product. It is difficult, maybe impossible, to give general advice on how to find these patterns, because domain-specific patterns are specific and not general.

There is no silver bullet to find patterns in the domain. One needs a thorough understanding. This is even true for ostensibly simple facts. For example, the German registrar has a view of persons' names that differs fundamentally from other public authorities. The registrar needs to disassemble names according to the role they play in civil law. He knows and distinguishes first name, family name, additional name, birth name, assumed name, title, and name affix. The residents' registration office, by comparison, just distinguishes first name, last name, and name affix. With respect to an address, the level of detail demanded by registrar and residents' registration office is the opposite way round: The residents' registration office distinguishes street name, house number, number of rear building, and floor. The registrar shows no interest in this detailed information. He just needs street name and house number.

As this little example shows, the analysis of the domain is the basis for finding patterns, for finding the vocabulary of the domain. Understanding the vocabulary of the domain is the precondition to find means to describe the domain thus, for a domain-specific language.

6.2. Characteristic Properties in the Domain

It is impossible to describe generic, domain-independent patterns, but it is possible to describe *properties* of the patterns one might look for. The designer of domain-specific languages has to observe the domain, i.e. the problem space, and has to discover the properties that the descriptions must capture.

We categorize the properties we incorporated in the domain-specific languages as follows:

- Given properties
- Anticipated properties
- Requested properties
- Designed properties
- Deduced properties
- Introduced properties.

In the following sections we discuss the properties of patterns that we found in our domain and used to design the domain-specific languages. We also give examples taken from the world of the registrars, their concepts, and their work practices.

6.2.1. Given Properties

The property is *given* by the domain, one has to understand it and support it in a convenient way.

A given property can be discovered by analyzing the domain. Sometimes comparison with other domains helps, because differences may be candidates for given properties.

Example: Registrars write text in fields on forms. This is true for many domains. But registrars must prevent subsequent insertion of text. So, only monospaced fonts are used, text starts immediately after any preprinted words, text ends with a final character, remaining white space is crossed out with a fill character, and printing on reserved positions is declared explicitly.

This is a given property, and the software must do the same. (Interestingly enough, Germany and Austria use different final characters.) These properties given by the domain strongly influenced the design of both the document description language and the `document formatter`.

6.2.2. Anticipated Properties

The *anticipated* property is not given, but will probably appear in the future. One had better provide support for this property.

Such a property can be found by thoroughly observing an aspect of the domain and comparing it with similar aspects in other domains. If more modern aspects appear in other domains, they might become relevant in this domain.

Example: Registrars fill out forms. Printing onto different preprinted forms is cumbersome. We expected that (at least a part of) the forms would be replaced with electronic forms sooner or later. We therefore included the possibility to print both form and text in one go. Years later, the legislation allowed electronic forms for certificates.

6.2.3. Requested Properties

The property is not given *per se*, but domain experts and users *request* to act in a certain manner. Even though several solutions are possible, they insist on one.

A requested property can be found by listening to domain experts and users. What they say may not make sense immediately. But, if some behavior or feature is recognized as important to them, it must be realized.

Example: The registrar checks the legal correctness of a transaction. What is legally correct and unambiguous can be expressed logically. The software provides the logical rules to check the transactions. Nevertheless, our domain experts requested the possibility to switch off all logical checks. It must remain the registrar's authority to decide what is correct and what is not, because he is responsible for doing the right thing.

A well-designed language provides features that allow such requests. Sometimes, different users or groups of users express contradictory requests. This may cause a variant in the product line.

6.2.4. Designed Properties

The property cannot be derived from the domain, it is not given, anticipated, or requested. Rather, the property is *designed* during the design of the software.

Transforming tasks from the problem space to a solution in the software system will confront us with choices solely because the solution is a software system. It offers different ways to do things. One has to invent an appropriate way. In this situation, a property should be designed. A designed property should be used consistently throughout the development of the solution. The domain-specific language should provide support for these properties.

Example: The series of masks often has optional masks, depending on the particular data (e.g. if the fiancés have children, masks for the children's names and addresses will be needed). There are basically two strategies to organize the workflow: 1. By default, show the set of mandatory masks for a transaction. Provide buttons to access optional masks. 2. Automatically adjust the set of masks to the current transaction data. Provide direct access to the mandatory masks. Both strategies are possible. The choice made impacts the design of the domain-specific language.

6.2.5. Deduced Properties

Software reduces the complexity of the real world by introducing structure and schema. Its metaphors create a virtual reality. It is possible to *deduce* properties from these metaphors that are consistent with the image the software induces in the users' minds.

Virtually, the domain is schematized according to a congruous idea. If the users catch on to the underlying metaphor, they will do the right thing automatically. Domain-specific languages should be designed with this in mind.

Example: From the organization of some registry offices, we adopted the notions of departments, categories, and civil transactions. We perceived the structure of the registrars' work and deduced it as a property: A department is a group of categories belonging together, and a civil transaction is an instance of a category (e.g. categories of the birth department are registration of a birth, legitimation of a child, and adoption.) We structured the transaction data in the database and in the user interface according to this deduced property. Easy to understand, easy to use. It had the effect of standardization.

6.2.6. Introduced Properties

Software is like a new world. An *introduced* property appears only because of the software. In the real world, there is no such thing.

A property will be introduced 1. if it is impossible to realize the software without it or 2. to allow to do things electronically that cannot be done in the real world or 3. to facilitate the users' work.

Example: We observed that registrars write certain data over and over again, e.g. the name of their own office and today's date, or nationalities, countries, languages. Also names of towns, streets, offices, and hospitals. When writing by hand or on a typewriter, there is no other way, but writing or typing them over and over again. Not so on a computer.

Our software reduced the registrars' workload by providing comfortable and convenient code-controlled text insertions. The mask description language contained statements to add different types of code-controlled text insertion to a field, and eventually this property was used in many more cases than we had planned.

7. CONCLUSION

Developing this architecture and these mechanisms required not only a deep and thorough understanding of the domain but also the discovery of *patterns* in the *domain.* The mechanisms used in this architecture fit these patterns. In particular, our data model matches the structure of the civil transactions. We took it as a basis to schematize the domain. 'Problem analysis takes you from the level of identifying the problem to the level of making the descriptions needed to solve it' according to Michael Jackson (2001).

We join domain expertise with software engineering. Domain experts directly contribute to software development: In the data references, they have found a powerful, efficient means of expression. As the owners of `Mask Descriptions`, `Document Descriptions`, and `Category Descriptions`, they have gained control over the domain-specific aspects of application development.

The analysis of the domain has led to languages that allow the description and specification of product behavior. Variant descriptions specify variants in the product line.

ACKNOWLEDGEMENT

We would like to thank Klaudia Metzner, the owner and executive director of the publishing house, for her sponsorship, favor, and passion, and for bringing in her outstanding domain knowledge. We thank the software development team at the publishing house, namely Thomas Stahl who co-designed and implemented the domain-specific languages, Manfred Benna and Rolf Deubel who implemented the first generation of the product line, Jens Puhle who is the most meticulous programmer we have ever met, Heike Schröder who developed the user interface components, Olaf Horn who knows everything about the base level technologies, Jörg Thiemer who bridges technical and domain knowledge, and Paul Rosenberg who is the perfect users' advocate.

Although both of us have been with other organizations for four years now, we continue to observe the development of the registrars' software with interest. We wish the team all the best.

REFERENCES

Aho AV, Sethi R, Ullman JD. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley: Reading, MA.

Bass L, Clements P, Kazman R. 2003. *Software Architecture in Practice*, 2nd edn. Addison-Wesley: Boston, MA.

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. 1996. *Pattern-oriented Software Architecture: A System of Patterns*, Vol. 1. John Wiley & Sons: New York.

Fowler M. 2003. *Patterns of Enterprise Application Architecture*. Addison-Wesley: Boston, MA.

Gamma E, Helm R, Johnson R, Vlissides J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Boston, MA.

Harel D. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* **8**: 231–274.

Holub AI. 1990. *Compiler Design in C*. Prentice-Hall: Englewood Cliffs, NJ.

Horrocks I. 1999. *Constructing the User Interface with Statecharts*. Addison-Wesley: Harlow, England.

Jackson M. 2001. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley: Harlow, England.

Keller F, Tabeling P, Apfelbacher R, Gröne B, Knöpfel A, Kugel R, Schmidt O. 2002. Improving knowledge transfer at the architectural level: concepts and notations. *Proceedings of the 2002 International Conference on Software Engineering Research and Practice*, Las Vegas 2002.

Knöpfel A. 2003. *FMC Quick Introduction*. Hasso Plattner Institute for Software Systems Engineering: Potsdam, Germany. http://fmc.hpi.uni-potsdam.de.

Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall: Englewood Cliffs, NJ.

Samek M. 2002. *Practical Statecharts in C/C++*. CMP Books, San Francisco, USA.

Schmitz H, Bornhofen H. (ed.). 2001. *Dienstanweisung für die Standesbeamten und ihre Aufsichtsbehörden*, 2nd edn. Verlag für Standesamtswesen: Frankfurt am Main.

Schmitz H, Bornhofen H. (ed.). 2003. *Personenstandsgesetz*, 10th edn. Verlag für Standesamtswesen: Frankfurt am Main.

Silberschatz A, Korth H, Sudarshan S. 2002. *Database System Concepts*, 4th edn. Mc-Graw-Hill: Boston, MA.

Thai TL, Lam H. 2001. *.NET Framework Essentials: Introducing the. NET Framework*. O'Reilly & Associates: Sebastopol, CA.

Whittaker JA, Thompson HH. 2004. *How to Break Software Security: Effective Techniques for Security Testing*. Addison-Wesley: Boston, MA.

Yoder JW, Johnson R. 2002. The adaptive object model architectural style. In *The Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02) at the World Computer Congress in Montreal 2002*, Bosch J, Gentleman M, Hofmeister C, Kuusela J. (eds). Kluwer Academic Publishers. http://www.joeyoder.com/papers/.