Segregating the Layers of Business Information Systems

An interface-based approach

Johannes Siedersleben[1], Gerhard Albers[2], Peter Fuchs[1], Johannes Weigend[1]
[1]*University of Applied Sciences of Rosenheim, Marienberger Str. 26, D-83024 Rosenheim, Germany; +49/8067/9122 (phone), +49/8067/9123 (fax)*
[2]*Software Design & Management, Thomas-Dehler-Str. 27, D-81737 Munich, Germany; +49/89/63812-0 (phone), +49/89/63812-150 (fax)*
johannes.siedersleben@t-online.de

Key words: Business information systems, external representations, interfaces, layers, reusability

Abstract: This paper presents a refined layered architecture for business information systems of any size. It allows a strict separation of application logic, database access, and user interface and is largely independent of programming languages, database management systems, operating systems, and middleware.

1. INTRODUCTION

1.1 Business information systems

Business information systems (e.g., systems for order processing, stock control, or flight reservation) are used daily by many people and are crucial for a company's business. The focus of this paper is front office systems as opposed to back office systems like data warehouses. The systems considered here can be characterised as follows:

– They are individually designed and implemented for big companies (telecommunication, railroad, travelling, car production). It takes more than one calendar year and several dozen man years to implement them. They contain at least several hundred thousand lines of code.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

- Their class (data) model contains several thousand attributes and several hundred classes (entities). They handle many gigabytes of data..
- They cope with high transaction rates, the transactions being short and relatively simple.
- They run in a heterogeneous environment: A system might involve several programming languages (e.g., Java, C++ and Cobol), several database management systems (e.g., SQL Server and DB2) and several operating systems (e.g., MVS, Unix and Windows NT).
- Their expected lifetime is 10 years or more.

In spite of many valuable results in the area of software engineering, the design of these kinds of systems has proved to be difficult. This paper presents a standard architecture, based on defined interfaces between components, that simplifies the design of these systems. It distils the experience of several dozen software projects in which the authors have been involved.

1.2 Why is software design so hard?

1.2.1 No metric for software design

We all know that software should be easy to maintain, easy to extend, easy to reuse, open to additional features, and fast. These properties are hard to measure (performance excepted), hard to achieve, and some of them are contradictory. A crucial feature like extensibility can at best be defined in terms of examples. There is no beaten path to a defined degree of maintainability, extensibility and so on; it all depends on the intuition of the system architect. The degree of maintainability actually reached by a given project is visible only after many years. It is not measured by quantitative means, but only assessed by the naked eye.

1.2.2 The three layer architecture does not work

The three layer architecture (see Ambler, 1998; Denert, 1991) is a well established recipe for the design of business information systems:

- the dialog layer controls the interaction with the user
- the application kernel implements the business logic
- the database access layer takes care of all database accesses

There has been little change to this scheme during the last few years; variants being discussed in the area of workflow systems do not affect the key ideas of this architecture. The intended benefit is the separation of concerns:

- The application kernel is neither aware of the user interface nor the database; changes to the these are transparent to the kernel.
- Dialog and database access layer have limited knowledge of the application. They are not aware of the business logic.

Experience shows that it is hard to keep the details of user interface and database off the application kernel. Two phenomena are frequently observed:

1. The business logic moves from the application kernel to other layers; the application kernel just vanishes.
2. The application kernel gets polluted with details that should be hidden in other layers.

There is a blatant lack of standards for the interaction of layers. Numerous projects have spent many years designing and redesigning these layers' responsibilities. To our experience, the intended benefit of the three layer approach never materialised to the expected degree.

1.2.3 Too many APIs

The software community is literally flooded with new technical APIs and new versions thereof: JDBC, ODBC, OCI, ADO, OLE-DB, AWT, MFC and so on. This makes the software architect's job even harder: Which API can I rely on? Which one works? How many workarounds will be necessary? How expensive would it be to migrate from – say – ODBC to OCI or vice versa? Furthermore there are some old-fashioned, awkward-to-use but very reliable host APIs (e.g., BMS, IMS, VSAM) that will not disappear in the near future and often have to be taken into account even with new systems. How can we cope with this variety of different APIs of different ages? For small systems with a short lifetime, these questions are of little importance. Our concern, however, is big systems with an expected lifetime of 10 years or more. These systems must carefully encapsulate all technical APIs.

1.2.4 Where to go?

Why are some components reusable and extendable, but others not? There is one obvious observation: Software that deals with many different things at a time is bad in all respects. The programmer's nightmare is return codes from different technical APIs mixed up with application problems, and all that within a couple of lines of code. This idea can be formalised: Any business information system is concerned with the application domain (this is why it is built), and technical APIs like operating systems, database management systems, and middleware (no system can run in thin air). Therefore, the components of a given system can be divided into four disjoint categories of reusability. Any piece of software can be:

1. determined neither by the application nor by technical APIs
2. determined by the application, but not by technical APIs,
3. determined by technical APIs, but not by the application,
4. determined by the application and by technical APIs.

The term " determined by" can be read as "knows about," "depends on," or "is influenced by." Code determined by the application knows about business objects like customers, accounts, flights, or aircraft. Code determined by technical APIs knows at least one API like ODBC or OCI. For the sake of convenience, we mark software determined by the application with an "A" and software determined by technical APIs with a "T", thus yielding the four categories 0 (neutral), A, T and AT.

0-software is ideally reusable, but of no use on its own. Class libraries dealing with strings and containers (e.g., STL) are examples for 0-software. 0-software implements an abstract concept, such as a dictionary or a state model. Note the difference between a class library like STL and a technical API like MFC that acts as an interface to a lower-level API (Win32). Using STL means choosing an abstract concept (namely that of containers, iterators, adapters, etc.) and works wherever C++ runs. Using MFC excludes all environments MFC does not support.

A-software can be reused whenever the given application logic is needed as a whole or in parts. Other applications access A-software typically via middleware like CORBA, DCOM or RMI.

T-software can be reused whenever a new system uses the same technical environment (e.g., JDBC, ODBC, AWT or MFC). One nice feature of T-software is that its size increases sub-linearly with respect to the number of business classes. A cleverly designed and carefully written technical component that works fine for 20 business classes can do as well for 200. In fact, JDBC (and other APIs as well) does not care at all about the number of business classes that are using it.

AT-software is hard to maintain, reluctant to change, can never be reused, and should hence be avoided. The architectural quality of a software system is inversely proportional to the share of AT-code. Unfortunately, at least at a small scale AT-code is easy and straightforward to write. Quality software is characterised by the complete lack of AT-code and by clean interfaces between 0, A and T. This is where we should go. It goes without saying that there are major management issues to the question of reusability that this technical paper does not address.

2. QUALITY SOFTWARE ARCHITECTURE

2.1 Overview

It is possible to define a standard architecture that contains some 0-components, no AT-components at all, and which establishes clean interfaces between 0, A and T components. This architecture is being developed by a project at Rosenheim University of Applied Sciences (Germany) in cooperation with Software Design & Management, a company in Munich. Its name is QUASAR, from "quality software architecture."

QUASAR employs the terms "use case" and "business object" in the sense of Jacobson (1997) with the following refinement: A use case seen as a software module knows which steps have to be performed in which order for the use case to succeed. A step of a use case can be any operation on business objects or on other use cases. A use case can be persistent (stay alive for days or months) or transient. There is no clear distinction between a use case and a business object. A flight reservation may be regarded as a use case, a business object, or both at a time; it is up to the designer to choose. QUASAR makes minimal assumptions about the design of use cases and business objects. QUASAR's concern are reusable components that are called by the application and which call it back.

QUASAR's mission is a standard architecture for business information systems that significantly simplifies the design. That means that there are reusable components of a defined category with defined interfaces, and running prototypes in evidence of feasibility that can be used as-is or as templates for project-specific implementations. The remainder of this paper describes the current state of our work.

2.2 Central themes

The central themes of the quality software architecture are

- QUASAR attempts to be as non-intrusive as possible. An application using QUASAR has to implement defined interfaces and to call others. All assumptions are laid down as interfaces. In particular, there is no superclass from which all business objects or use cases have to be derived.
- QUASAR is open to almost all programming languages. There is a focus on object-oriented languages, but QUASAR components can be implemented in C or Cobol as well. QUASAR doesn't rely on language features like RTTI (C++), Java reflection classes or Smalltalk blocks.
- QUASAR shields the application kernel from technical APIs (like OCI or ODBC) by means of a stable, vendor-independent interface.

- QUASAR components can be used independently.
- QUASAR avoids code generation. Often code generators tend to be slow, unreliable, do not generate what is really needed and turn out to be a burden for the development process.
- QUASAR only performs error and exception detection. The handling is left to the application.

2.3 Architecture

QUASAR's backbone are business objects and use cases. The main menu of most systems can be thought of as a special use case which gains control at system start-up. We call this the use case controller. Normally, a use case is started by its constructor or by a special start method.

On the right hand side of Figure1 there are three components: The **Workspace**, the **DataStore** and the **concrete API,** which provides access to the database (ODBC, OCI,..). The DataStore hides that API behind a generic interface which can be talked to in terms of DataContainers. Thus, it is unaware of use cases and business objects. The DataStore interface provides the usual find/update/insert/delete operations.

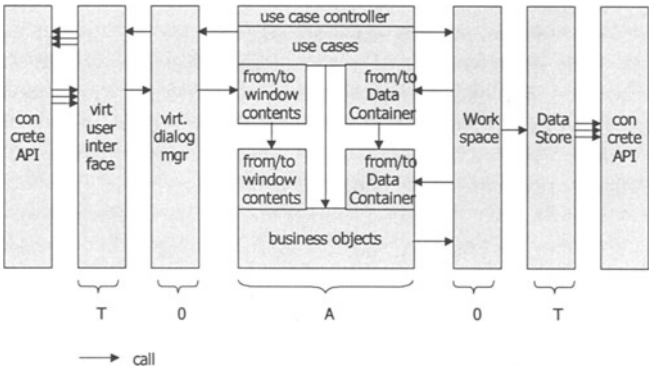

Figure 1: QUASAR architecture

We use the term "persistent object" for all business objects and all use cases that are to be stored in the database. In general, all business objects and some of the use cases will be persistent. A persistent object cannot be stored as such, but only as a DataContainer. So, each persistent class has to provide methods to map the object onto its representation as a Data-Container and vice versa. In most languages, these methods will have names

like "toDataContainer" and "fromDataContainer"; in C++ it is simple to overload the shift operators.

Following Java naming conventions, the interface that defines these methods is called "Storable"; all persistent classes must implement it. In section 2.6, we sketch how this can be done. The Workspace links the application to the DataStore. Its interface follows closely that of DataStore, but is defined in terms of Storables. Thus, it is the Workspace that calls the Storables' mapping methods. The Workspace takes care of object identity and implements a given transaction strategy (optimistic or pessimistic, see section 2.6). The important thing to note is that the Workspace is 0-software. Its use could become as obvious as that of – say – a container. The communication between the application and the database uniquely relies on two interfaces: The Storable interface with its to- and from- methods and the Workspace interface. This is the only link between the two worlds; there are no assumptions about each other except those cast in the two interfaces.

In a typical implementation, there will be exactly one instance of Workspace and one instance of DataStore for each human user logged in. Variants of this rule are hinted at in section 2.6.

Let's look at the left hand side of the figure. There is a symmetry not only in the figure but in the whole way of thinking. We will see that accessing a database and accessing a user interface have a lot in common.

Again there are three components: The **virtual dialog manager** (VDM), the **virtual user interface** (VUI), and the **concrete API**, that provides access to the physical screen, which can be anything in the area of BMS, Motif and MFC. The virtual user interface hides that API behind a generic interface that can be talked to in terms of virtual windows and virtual widgets. Like the DataStore, the VUI is unaware of use cases and business objects and knows basically only two classes: virtual windows and virtual widgets.

We use the term "presentable object" for all business objects and all use cases that are to be presented to the user. In general, all use cases and most of the business objects will be presentable. A presentable object cannot be presented as such, but only as a virtual window. In complete analogy to the database side, each presentable class provides the methods "toVirtual-Window" and "fromVirtualWindow". Of course, the corresponding interface is called "Presentable". The virtual user interface presents virtual windows by means of its central method "processVirtualWindow". Within that method, it handles incoming events. Many of them can be dealt with directly by the VUI, for example, field editing. The main benefit of the VUI is its ability to condense physical events (e.g., "button X released", "field Y changed") into virtual events. Virtual events are abstractions of physical events, e.g., "analyse user input", "confirm" or "cancel". In the simplest case, a physical event is directly mapped onto one virtual event. In general,

there can be arbitrary definitions like "button Y released and field Z changed and field T not changed". This idea works for graphical user interfaces as well as for block-oriented ones: In a 3270 environment, there are very few physical events ("key *K* is hit, where *K* is one of ENTER, PF1, PF2,...") which can be easily refined to many different virtual events. Whenever the virtual user interface recognises a virtual event it calls back the virtual dialog manager and tells it to process the virtual event.

The virtual dialog manager acts as a link between the use case that wants to execute its dialog and the VUI that communicates with the concrete API. Each use case has a corresponding instance of VDM, which in turn has an instance of a VUI. Thus, for each active use case, there will be one VDM instance and one VUI instance. Each VDM instance manages one dialog only and dies when that dialog is closed. The dialog management is controlled by an interaction diagram. Interaction diagrams have been used for quite a while and have proven to be extremely useful for the precise description of user interactions; see Denert (1991) for more details. Here, the use case hands over an instance of an interaction diagram to its VDM instance. Thus, the VDM constructor expects a presentable object (i.e., the use case) and an interaction diagram as arguments. It calls the use case's toVirtualWindow-method and transmits the result to its VUI. When the VDM instance is called back with the processVirtualEvent method, it consults its interaction diagram and decides what to do. Frequently the ruling use case is called back with its fromVirtualWindow method.

The VDM is little more than an interpreter of interaction diagrams and fairly easy to implement. It is, of course, 0-software. The to/fromVirtual-Window methods are far less straightforward; see section 2.7.

An active database would call back the DataStore very much like BMS/CICS or Motif would call back the VUI. Both the VUI and VDM can be hierarchically organised along the lines of the PAC pattern (Bass, Coutaz, 1991).

2.4 Virtual devices

Virtual devices encapsulate technical APIs. We have seen two of them:

1. the virtual user interface (VUI), hiding APIs like BMS or MFC
2. the DataStore, hiding, for example, ODBC or OCI

Additional virtual devices can be introduced for any technical API that should be hidden from the system (e.g., workflow systems or archives). Virtual devices don't know anything about customers, accounts and orders; instead, they deal with virtual containers containing virtual items. VUI is concerned with virtual windows and virtual widgets; DataStore manages DataContainers and DataContainerColumns. The definition of a virtual

container consists of three parts: definition, contents, and context, which can be implemented as a class or a record each, depending on the programming language. In order to bring a virtual device into being, you define the interface, the item, the container, and you implement the interface for at least one concrete API. This is, of course, a T-implementation (determined by exactly one API). It is written in the same language as the API, that is, a DataStore implementation for ODBC is likely to be written in C or C++; a JDBC implementation in Java. A virtual device only knows a handful of classes.

It is up to the designer to determine the amount of information known by virtual devices. Choosing a complex virtual container allows full exploitation of the features of the physical device but makes the from/toVirtualContainer methods expensive to implement and reduces portability. Choosing a dumb virtual container guarantees portability, allows for cheap to/from-methods, but a 3270-minded virtual window won't look very beautiful on a Motif screen. This is not as harsh a problem as it might appear at first sight: Many famous standard products (e.g., SAP) have a graphical user interface that is almost completely form based and could be implemented by means of a rather dumb virtual window.

2.5 DataStore

This section describes the DataStore interface and its implementation with a relational database in mind. However, it is equally possible to have the interface implemented for an object-oriented database or for VSAM. Why would somebody choose not to directly access an object-oriented database but rather via a DataStore? There could be at least two reasons:

1. In spite of the ODMG efforts, the available object-oriented database management systems differ significantly. Many vendors are small companies whose future is hard to predict. Within the context of a reengineering and/or migration project, a given application may switch from a relational database to an object-oriented one, or, even worse, may have to access both of them at a time. So, for many applications it is crucial to separate database-influenced code from application logic.
2. Even with object-oriented databases, database classes and application classes are not necessarily identical. Performance considerations at the database level should be invisible at the application level.

It depends on the information conveyed by the DataContainer if the DataStore-implementation is able to exploit the actual database's features.

Let's look at a relational DataContainer, that can be mapped onto one or more physical rows of one or more tables. The **DataContainerDefinition** is a data structure (class or record) that contains all information necessary to

talk to the database about rows of a particular table (field name, data type, length, precision, primary keys, etc.). It is up to the software architect to allow few or many data types in the definition. The DataStore maps these virtual data types onto the physical ones.

The **DataContainerContents** is a container of column contents that in turn are just values of the corresponding data type. It contains a reference to the corresponding DataContainerDefinition that tells the DataStore how to read the contents. It is crucial to make sure that a given contents matches the definition it refers to.

The **DataContainerContext** contains additional information for the DataStore about how to process given contents. When writing to the database, it can be important to know which fields are unchanged; when reading, perhaps not all fields are requested. DataContainerContext is a container of column contexts that convey state information such as "changed/unchanged" or "requested/not requested".

The DataStore interface defines the ordinary database operations in terms of DataContainer definition, contents, and context. The most obvious operations are:

```
DsReturnCode find(DataContainer dc) throws DsException;
void update(DataContainer dc) throws DsException;
void insert(DataContainer dc) throws DsException;
void delete(DataContainer dc) throws DsException;
```

The DataStore also supports the database's transaction logic:

```
DsReturnCode commit() throws DsException;
void rollback() throws DsException;
```

It depends on the chosen transaction strategy if we need an additional operation for locking:

```
DsReturnCode lock(DataContainer dc) throws DsException;
```

The DataStore's return codes (class DsReturnCode) are used for normal events (e.g., find() didn't find anything, commit() encountered a collision with another user); exceptions are raised for unexpected events (e.g., database not available). The DataStore interface provides at least one operation for bulk reading.

```
DsResultSet findMany(DataContainer dc) throws DsException;
```

This is a query by example: findMany accepts an example and searches all matching DataContainers. The DataContainerContext tells the DataStore which fields are to be read from the database. This operation returns an instance of DsResultSet, that implements the usual Collection interface, but can be finely tuned with respect to prefetching and caching. It is accessed by an iterator; the actual database fetch operation may happen at any time between the invocation of findMany (earliest possible point in time) and the dereferencing of the iterator (latest possible point in time). A variety of similar findMany methods can be implemented accepting more than one example connected by logical expressions. Experience shows that almost all queries of a typical OLTP application can be dealt with in this manner. The DataStore also supplies DDL-methods, so it can check at runtime whether the actual database layout matches the actual classes.

The DataStore sketched here is easy to implement, especially when it can be copied from a template. However, it does not provide the full query functionality of SQL or OQL. If this is required there is an obvious work-around: a findMany() method that directly accepts an SQL or OQL query string. However, this pollutes the application kernel, effectively transforming it into AT-software, so the workaround should only be a well documented, rarely used hack.

2.6 Workspaces and storables

The Storable interface is implemented by each persistent class:

```
void toDataContainer(DataContainer dc);
void fromDataContainer(DataContainer dc);
void resolve(Workspace ws, DataContainer dc);
Oid getOid ();
Storable clone();
```

The to/from methods have been discussed already. They are easy to program: The to/from methods of a complex class call the to/from methods of its components; the DataContainer itself knows how to handle elementary data types (`int`, `float`, and so on). This is an application of the well-known streams concept that is used similarly by, for example, XDR (external data representation) or NDR (network data representation). The to/from-methods map the object onto its database representation and vice versa. For example, several object fields may be combined into one database field. This is why in general it is not a good idea to have these methods generated.

The resolve method resolves the object's references: It knows the foreign keys contained in the DataContainer and calls the Workspace's find-method in order to get an object reference or a container of references. The clone-method is needed for technical reasons.

Inheritance is easily dealt with if the following rules are observed:

- There is one DataContainer for each persistent object regardless of the number of superclasses contributing to that object. The DataContainer contains a discriminator indicating the actual class.
- The toDataContainer method of any derived class first calls the toDataContainer method of its superclass (like a constructor).
- The fromDataContainer method of any superclass calls the fromDataContainer method of the actual derived class using a switch-statement on a discriminator contained in the DataContainer.

It is the DataStore's job to map the DataContainer according to the DataContainerDefinition onto one huge table (one table per inheritance tree), many tables (one table per class) or anything in between.

The Workspace interface is almost identical to the DataStore interface except that Workspace deals with Storables whereas DataStore only knows DataContainers. If we implement DataContainers as Storables, then any Workspace implementation automatically implements DataStore and can be used anywhere a DataStore is expected.

The Workspace's primary task is to call the appropriate to/from and resolve methods. It has, however, further reasons for existing:

- It implements object identity, that is, subsequent finds yielding the same object return a reference to that object, not a copy. The Workspace needs the getOid() method in order to identify objects.
- It implements a given transaction strategy. The optimistic strategy reads without lock and checks only at update time if there was a collision with a different user; the pessimistic strategy locks all objects on read.

The idea of the Workspace is "What you say is what you get." All changes of objects sharing the same Workspace are immediately visible to all those objects that represent together an area of integrity. It is only at commit time that these changes are published to the DataStore behind the Workspace. Pursuing this idea a bit further, one can imagine an arbitrary tree of Workspaces, each managing an integrity area and communicating by means of the publisher/subscriber pattern.

Another extension of Workspaces could handle (not implement!) the 2-phase-commit protocol. A Workspace could have several DataStores hiding different databases. The Workspace commit would be translated into the well-known prepare-to-commit/commit loops . This works fine and is easy to implement if all databases involved understand that protocol. The benefit is that the application is not aware of anything like a 2-phase-commit.

A Workspace should be written in the same programming language as the application and should be directly linked to it. A client/server cut between application and Workspace can be compared to accessing the STL by – say – CORBA. But a client/server cut between Workspace and DataStore is a very natural matter. It is perfectly possible to have the application written in Java and to provide fast access to Oracle by a DataStore written in C and using OCI. All you have to do is to transform DataContainers between Java and C. There are many ways of doing that using sockets, JNI, RMI, CORBA, or combinations thereof.

2.7 Virtual user interface (VUI)

The virtual window contains information about things to be displayed. A virtual window can be displayed as (part of) a physical window or distributed among several physical windows. Depending on the concrete API, the VUI provides callback methods for all physical events it understands. It is up to the designer to make virtual windows very intelligent (e.g., know about things like tree views) or rather dumb (e.g., 3270-based). At any rate the virtual window contains:

- The definition of virtual events. It is the VUI's main job to map physical events upon virtual ones thus keeping the other components free of knowledge about screen, mouse, keyboard, and other devices.
- The data types of the input fields. The VUI performs all field related type checks. The more data types a given implementation knows the more checks it can perform. If a given VUI implementation encounters a data type it doesn't know, it calls back its VDM.

The central method of the VUI is "processVirtualWindow". Within this method, the VUI listens to physical events until it recognises a virtual event in which case the VDM is called back. There is one instance of VUI for each active dialog.

The mapping between an object and its representation as a virtual window is similar to the mapping between an object and its representation as a DataContainer. For example, it is up to the VUI to decide in function of the screen size to represent a given virtual window as one or more physical windows (that is, a window provided by the concrete API).

2.8 Virtual dialog manager (VDM)

The virtual dialog manager is instantiated every time a use case decides to execute itself. Its main ability is to present Presentables (objects implementing the to/fromVirtualWindow-methods) and to process virtual events when being called back by its VUI-instance.

It manages the dialog by means of an IAD (interaction diagram). This is a finite state machine that controls the states of a dialog. For each state, there is a set of legal virtual events. The IAD indicates which action is to be executed when a given virtual event occurs and it defines the resulting state in function of the outcome of that action. These actions may be known to the VDM itself (e.g., close_window) or they are methods of the ruling use case. The use case has to register these methods with the IAD. In general, most of the dialogs will be covered by just a handful of standard IADs.

2.9 Virtual windows

A common argument against this kind of architecture says that implementing virtual windows amounts to reimplementing the widget hierarchy of AWT, Motif, or whatever. This can be avoided by using concrete widgets directly within virtual windows, thus accepting use cases that are no longer A but AT. When designing virtual windows, it is important to know whether or not there is a binary link between use cases and VUI. If so, the field's data types, for example, can be simply given as interfaces the VUI calls back each time a field is edited. Likewise, the concrete widget classes can be used directly by the use case. If not, all information in the virtual window has to be coded as strings interpreted by the VUI. This latter choice is obviously suboptimal as far as performance is concerned, but it is ideally suited for a client/server cut between VUI and VDM. For example, it is possible to have a VUI implemented as an applet talking to a remote application (including VDM, use cases, business objects) written in any language into which the string based virtual window can be translated.

3. BENEFITS

Let us summarise the main benefits of QUASAR:

- Virtual devices only know about virtual containers. Hence, it is very convenient to have a client/server cut between a virtual device and the remainder of the application. The IDL contains only a handful of class definitions (the virtual container and its items). As a general rule you shouldn't have many business objects on both sides of a client server cut: Maintaining consistency can be a nightmare, even with CORBA.
- It is not hard (even without CORBA) to translate a virtual container from one programming language to another. This is obvious for Java and C++, for example, but can also be done between C++ and COBOL. It is possible to have a VUI written as a Java applet talking to an application

written in C++ talking to a DB2 database via a DataStore written in COBOL.
- Virtual containers can be dumb or intelligent. A dumb container can easily be mapped onto an intelligent one; the other direction is harder, but often possible (an OK-button can be represented as a yes/no input field). Virtual containers could be standardised: The software community doesn't need more than two or three of each kind.
- Implementation of use cases and business objects is not affected by any technical API. There is a direct transformation from the application class model (given in – say – UML notation) to the implementation classes.
- The database design determines the to- and fromDataContainer methods and nothing else. Any change of the database layout only affects these mapping methods.
- The user interface design determines the to- and fromVirtualWindow methods and nothing else. Any change of the windows layout only affects these mapping methods.

There are two important points beyond the QUASAR story:

1. We, the community of software designers, badly need well-defined interfaces between the layers of the classical architecture or variants thereof. Every working day there are many thousand software designer thinking about basically the same design problems. There **must** be an answer to that!
2. Sooner or later the tremendous, unfiltered amount of new technical components will drive us crazy. There **must** be a way to enjoy new features without being forced to migrate complete systems from Java 1.0 to 1.1 to 1.2 to 1.x or from RDO to ADO to OLE-DB or to whatever is cool next week.

REFERENCES

Ambler, Scott W. (1998), Building Object Applications That Work, Cambridge University Press & SIGS Books

Bass, L., J. Coutaz (1991), Developing Software for the User Interface, SEI Series in Software Engineering

Denert, E. (1991), *Software Engineering*, Springer Verlag.

Heuer, A. (1997), *Objektorientierte Datenbanken*, Addison Wesley.

Jacobson, I. , M. Griss, P. Jonsson (1997), *Software Reuse*, Addison Wesley.