1. ABSTRACT
This paper presents a refined architecture for business information systems of any size. It allows for 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.

1.1 Keywords
Generic Architectures, Domain-specific Architectures, Architectural Styles and Patterns

2. INTRODUCTION
This paper deals with business information systems which are used daily by many people and which are crucial for the company's business. Examples are systems for order processing, stock keeping, flight reservation and so forth. The focus of this paper are front office systems as opposed to back office systems like data warehouses. The systems considered here can be characterized 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 those systems. They contain several hundred thousand lines of code, sometimes several millions.
- The 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 an heterogeneous environment: One 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).
- The expected lifetime is 10 years or more.

The design of this kind of systems has shown to be extremely difficult.

3. WHY IS SOFTWARE DESIGN SO HARD?

3.1 There is no Metric for Software Design
We all know that software should be easy to maintain, easy to extend, easy to reuse, open to new or additional features (e.g. a new applet-based user interface) and fast. These properties are hard to measure (performance excepted), hard to achieve and some of them are contradictory. A crucial feature like extendibility can at best be defined in terms of examples. There is no beaten path to a defined degree of maintainability, extendibility and so on. All depends on the intuition of the system architect. The degree of maintainability or extendibility actually reached by a given project is visible only after many years. It is not measured by quantitative means, but only assessed with the naked eye.

3.2 The Three Layer Architecture Does not Work

The three layer architecture is a well established recipe for the design of business information systems:

- There is a dialogue layer which controls the interaction with the user;
- the application kernel implements the business logic;
- the database access layer takes care of all database accesses.

See Denert [1] for more details. There has been little change to that simple 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 of the database. Hence, changes to user interface and/or database (layout and/or DBMS) should be transparent to the application kernel.

The dialogue and database access have a limited knowledge of the application. They are not aware of the business logic.
However, experience shows that it is extremely hard to keep the details of user interface and database off the application kernel. There are two phenomena which are frequently observed:

- The business logic moves from the application kernel to other layers; the application kernel just vanishes, or:
- The application kernel stays alive, but gets polluted with details which should be hidden in other layers.

In fact, there is a blatant lack of standards for the interaction of layers. Numerous projects have spent many man years with designing and redesigning these layer’s responsibilities. To this author’s experience, the intended benefit of the three layer approach never got materialized to the expected degree.

3.3 There are 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? For small systems with a short lifetime, these questions are of little importance. Our concern, however, are big application systems with an expected lifetime of 10 years or more. It is mandatory for these systems to carefully encapsulate all technical APIs.

4. WHERE TO GO?

Why is a given component well reusable, extendable and another one is 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 are return codes from different technical APIs mixed up with application problems and all that within a couple of lines of code. Let us formalize this idea.

Any business information system is concerned with

- the application: this is why it is built, and
- technical APIs (operating system, database management system, middleware), because no system can run in thin air.

Therefore, the components of a given system can be divided up into four disjoint categories of reusability. Any piece of software can be:

- determined neither by the application nor by technical APIs
- determined by the application, but not by technical APIs,
- determined by technical APIs, but not by the application,
- determined by the application and by technical APIs.

Here, the term "determined by" can also be read as "knows about", "depends on" and "is influenced by". Code determined by the application knows about specific things 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 call software belonging to category $X$ simply $C_X$ software, where $X$ runs from 1 to 4.

- **C1 software** is ideally reusable, but it is of no use on its own. Class libraries dealing with strings and containers (e.g. STL) are examples for C1 software. C1 software always implements an abstract concept, e.g. that of a dictionary or a state model.

- **C2 software** can be reused whenever the given application logic is needed as a whole or in parts. Other applications access C2 software typically via middleware like CORBA, DCOM or RMI.

- **C3 software** can be reused whenever a new system uses the same technical environment. One nice feature of C3 software is that its size increases sublinearly with respect to the number business classes. A cleverly designed and carefully written technical component that works fine for 20 business classes can do as well for 200.

- **C4 software** is hard to maintain, reluctant to change, can never be reused, and should hence be avoided. The share of C4 code (measured as a quotient of lines of code) is inversely proportional to the architectural quality of a software system. Unfortunately, C4 code is easy and straightforward to write, at least at a small scale.

So, quality software is characterized by the complete lack of C4 code and by clean interfaces between C1, C2 and C3. This is where we should go.

Additional Remark: It goes without saying that there are major management issues to the question of reusability which this technical paper does not address.

5. QUASAR

It is possible to define a standard architecture which contains some C1 components, no C4 at all and which establishes clean interfaces between C1, C2 and C3 software. This architecture is being developed by a project at Rosenheim University of Applied Sciences (Germany) in collaboration with software design & management, a software company in Munich, Germany. Its name is "Quasar" which stands for "quality software architecture".

We give a sketch of Quasar’s key ideas.

Quasar employs the terms "use case" and "business object" in the sense of Jacobson [2] with the following refinement: A use case seen as a software module (e.g. a class) knows which steps have to be performed in which order for that 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 (that is, it stays alive for days or months) or not persistent. Quasar makes minimal assumptions about the design of use cases and business objects. Quasar’s concern are the following reusable
components which are called by business objects and use cases and which call them back.

5.1 Virtual Devices (C3) and Virtual Containers (C1)
Each concrete API can be encapsulated by a virtual device handling virtual containers. The virtual database (VDB) hides APIs like OCI (Oracle call interface) or ODBC (open database connectivity); its virtual container is a virtual row containing virtual fields. A virtual row is made up of its structure, its contents and its context. The structure (a class with few instances) contains all the information needed to map row contents (a class with many instances) onto one or more physical rows. The context may convey additional information (e.g. the fields to be updated, the fields to be read from the physical database). The same idea works fine for the user interface: The virtual user interface (VUI) hides APIs like BMS (CICS basic mapping support) or MFC (Microsoft Foundation Classes); it is concerned with virtual windows and virtual widgets. Virtual devices are interfaces (in the sense of Java or CORBA). There will be one C3-implementation for each concrete API that must be met. Virtual devices can be thought of as interpreters for virtual containers. They can be designed for other technical APIs as well (e.g. that of a workflow system).

5.2 Transformation Methods (C2)
Objects which are to be stored in the database cannot be stored as such, but only as a virtual row. So each storable class has to provide two methods for the transformation between the object itself and its representation as a virtual row. The interface which defines these methods is called "Storable". So, the communication between the database and the application uniquely relies on two interfaces, namely Storable and VDB. Likewise, all presentable objects implement the interface "Presentable": They transform themselves into a virtual window and reconstruct themselves from that virtual window. These to- and from-methods reflect the database design (in the case of storables) and the screen design (in the case of presentables).

5.3 Virtual Transaction (C1)
The virtual transaction is a C1-component which sits between the virtual database and the application. It is the virtual transaction which calls the transformation methods of the business objects. The virtual transaction takes care of object identity and implements a given transaction strategy (e.g. optimistic or pessimistic). In general, there will be one instance of a virtual transaction and of a virtual database for each user logged in. Please note that in any case, both are strictly single user modules. Any kind of concurrency between different users should be left to dedicated software, e.g. database management system and/or transaction monitor the use of which would be hidden by the virtual transaction.

5.4 Virtual Dialog Manager (C1)
The virtual dialog manager sits between the virtual user interface and the application. It controls the dialog states by means of interaction state models. See Denert [1] for more details.

6. Benefits
Let us summarize 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. In the IDL-file there are only a handful of class definitions (namely the virtual container and its contents). As a general rule it is not recommended to have too many business objects on both sides of a client server cut. Consistency between both sides can become a nightmare, even with CORBA.

- It is not hard (even without CORBA) to translate a virtual container from one programming language into another. This should be obvious for e.g. Java and C++, but it can also be done between e.g. C++ and Cobol. It is perfectly possible to have a VUI written as a Java applet talking to an application written in C++ calling to a DB2 database via a VDB written in Cobol.

- Virtual containers can be dumb (3270 oriented) or intelligent (know about 3D representations). A dumb container can easily be mapped onto an intelligent one: the other direction is less easy, 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.

- The implementation of use cases and business objects is not affected by any technical API. So, there is a direct transformation from the class model (in – say – UML notation) to the classes to be implemented.

- The database design determines the to- and fromVirtualRowMethods and nothing else. So, any change of the database layout only affects these transformations methods.

However, there are two even more important points:

- 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!

- Sooner or later, the tremendous, unfiltered amount of new technical components we are daily confronted with will drive us crazy. There must be a way to enjoy really new features without having 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 to XXX.

Quasar tries to contribute to these issues.
7. ACKNOWLEDGMENTS
My thanks to software design & management, Munich, Germany, for supporting and encouraging the QUASAR project.

8. REFERENCES