Goodchild, M.F., M.J. Egenhofer, R. Fegeas, and C.A. Kottman, editors.
Interoperating Geographic Information Systems. Boston: Kluwer Academic Publishers, 1999

1 GEOTOOLKIT: OPENING THE ACCESS TO OBJECT-ORIENTED GEO-DATA STORES

Oleg Balovnev, Martin Breunig, Armin B. Cremers, Serge Shumilov
Institute of Computer Science III, University of Bonn, Germany

1.1 Introduction

Today a typical geo-information system (GIS) is a complex „historically grown" software package which inevitably inherits the software-engineering practices of the past. Application programming within such systems is extremely complicated. Data structures and functions are often completely hidden from the user. As a result, they are hardly extensible to meet the requirements imposed, for instance, by 3D/4D-modeling. The next generation GIS should benefit from modern software engineering technologies among which one of the most promising is component-based design. Software building organized in libraries with consistent programming interfaces will enable the fast assembly of a special-purpose application for a particular domain. An application-specific component can be customized and re-used (with necessary extensions and modifications, if required) for the development of related applications. Following this approach, a general-purpose geo-information system will be substituted by a family of specialized sub-systems which, due to the common design basis, are well suited for the inter-communication and mutual data exchange.

An object-oriented programming environment can provide the necessary support required for the integration of diverse software components. In general, we are strongly convinced that object-orientation (and object-oriented data management in particular) will play a key role in the development of the future generation geo-scientific systems because it has the tendency to dissolve the boundaries between programs and databases, allowing data structures used in, e.g. numerical models, to be persistent objects, maintained directly by object-oriented database management systems. Complex data structures optimized for efficient computations in main memory need no longer to be pre-assembled from numerous relational tables. An integrated representation for algorithms and data within object-oriented databases enables access not only to the data but also to the data processing methods associated with the data. Due to this fact, object-oriented geodata stores could play the central role in data and service exchange between heterogeneous geo-scientific applications: uniformly stored database objects can serve as mediators between diverse application-specific representations. However, direct database-level interoperation is not always possible because of the extreme heterogeneity of already existing applications, software environments and hardware platforms. A necessary infrastructure to deal with the heterogeneity and remote access in the object-oriented context can be provided by a Common Object Request Broker Architecture (CORBA).

In this chapter we present our experience with *GeoToolKit* - a component software for the development of 3D/4D geo-scientific applications. Its backbone is a class library for the efficient storage and retrieval of spatial objects within an object-oriented database. Common data types inherited by diverse applications from the *GeoToolKit* spatial class hierarchy establish favorable conditions for the database-level integration of heterogeneous geo-scientific data. The key for the data integration is a common object model built on top of general spatial object class hierarchy provided by *GeoToolKit*. *GeoToolKit* has been successfully used in the development of several geo-scientific applications. We briefly introduce two of them, *GeoStore* and *GeoDeform* which interoperate one with another via a shared geo-data store. In conclusion we present a CORBA-based technique for opening geodata stores to external applications.

Goodchild, M.F., M.J. Egenhofer, R. Fegeas, and C.A. Kottman, editors.
Interoperating Geographic Information Systems. Boston: Kluwer Academic Publishers, 1999

1.2 GeoToolKit

Within the collaborative research center SFB350 at the University of Bonn we have developed a component software called *GeoToolKit* (Balovnev *et al*. 1997) which is intended to facilitate the design and implementation of 3D/4D geo-applications. The idea is to provide for an application developer a set of geo-oriented software building blocks involving DBMS-based spatial data maintenance, special support for efficient spatial retrieval, visualization, graphical interfaces and communication which the user may assemble in ready-to-use applications. *GeoToolKit* is not a closed GIS-in-a-box package - it is rather a library of C++ classes that allows the incorporation of spatial functionality within an application under development. It is primarily oriented on software engineers with C++ experience involved in the development of geo-applications for tasks which can be hardly performed within general-purpose GISs. At the same time *GeoToolKit* is not restricted to the class library. It also assumes a certain design methodology evolved on the basis of the experience we gained during the development of diverse applications on top of *GeoToolKit*. Some interactive tools under development involving a universal spatial data browser are expected to make GeoToolKit-based data stores available for the geo-scientists without the mediation of software engineers.

A special attention we paid to the generality and extensibility of the class library so that it could be easily adopted for diverse requirements. At the same time it was obvious for us that a toolkit should be more than just an empty interface specification, i.e. it should do something. *GeoToolKit* addresses primarily the efficient storage and access of 3D-spatial objects within a database. To achieve this *GeoToolKit* is tightly coupled with the object-oriented DBMS *ObjectStore*®. We did this in spite of potential disadvantages of the binding to the concrete DBMS: integration with still non-standardized object-oriented DBMS may demand a cardinal re-design of both interfaces and function bodies. We preferred from the very beginning to benefit maximally from *ObjectStore's* specific features, e.g. we intensively exploited its low-level clustering control facilities while developing DBMS-maintained spatial indexes. Though *GeoToolKit* is built on top of *ObjectStore* it is organized in such a way that its users do not need to know anything special about *ObjectStore*: all necessary data management facilities are incorporated in the *GeoToolKit* classes.

Currently *GeoToolKit* offers classes for the representation and manipulation of simple (point, segment, triangle, tetrahedron) and complex (curve, surface, solid) 3D spatial objects. We consciously avoided 2D-issues as far as it was possible. Following the object-oriented modeling technique, not only abstract geometric primitives, such as points, curves, and surfaces but real world entities such as drilling wells, geological sections or strata, can be modeled and maintained. Applications developed with *GeoToolKit* simply inherit geometric functionality from *GeoToolKit*, extending it with application-specific semantics.

1.2.1 Class Hierarchy

GeoToolKit is primarily oriented for managing real-world entities modeled as bodies with arbitrary complex shapes. However, such objects as curves, surfaces, lines and planes (though they do not correspond to any real-world object) also need to be maintained in *GeoToolKit* since these very useful geometric abstractions are utilized in various modeling and analyzing tools. Fig.1 presents their relationships within the data model of *GeoToolKit* shown in OMT-like notation (Rumbaugh *et al.* 1991).

Goodchild, M.F., M.J. Egenhofer, R. Fegeas, and C.A. Kottman, editors.
Interoperating Geographic Information Systems. Boston: Kluwer Academic Publishers, 1999

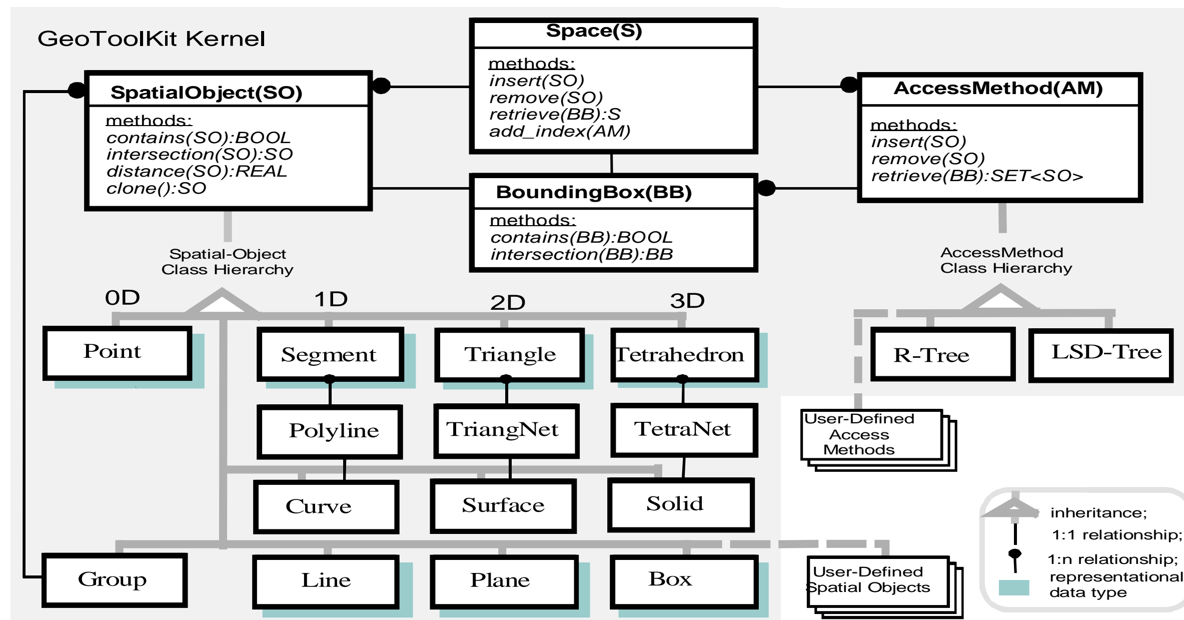

Fig. 1 *GeoToolKit*'s object model

The abstract *SpatialObject* class specifies the interface which is to be inherited by all concrete spatial objects. Any concrete spatial object must be defined as a specialization of the abstract *SpatialObject* class. A concrete class provides an appropriate representation for the object as well as the representation dependent implementation for the geometric functions. For example, since a minimal bounding box is permanently used for efficient computations, the *SpatialObject* class declares a function which returns a bounding box of the object. This function must be redefined and implemented in every direct specialization of the *SpatialObject* class. Thus *GeoToolKit* guarantees that every spatial object has at least the functionality of the most general class. Every particular spatial object class can additionally have geometric functions characteristic to only that class.

A real-world object usually containing additional non-spatial data is easily modeled as a specialization of a concrete spatial object, complemented with data members and methods provided by the application developer. Following this approach, a user-defined class automatically inherits the whole geometric functionality of its geometric parent. Sometimes it may be beneficial to introduce a completely new type. Then, instead of re-using *GeoToolKit*'s types, an application developer creates his own class as a direct specialization of the abstract *SpatialObject* class. In this case he is responsible for providing the implementation of the geometric functionality specified in the root class. Newly defined data types can be included in the *GeoToolKit* class library for the further re-utilization.

The *GeoToolKit* class hierarchy is complete: any spatial object can be modeled either directly by one of the built-in spatial classes or as a composition of these classes within a group. A *Group* is a heterogeneous collection of spatial objects which are further treated as a single object. It may contain simple and complex objects, as well as objects of different dimensions including other groups. Operations applied to a group are simply forwarded to separate objects contained in the group. Object grouping enables the organization of data in high-level abstractions without any loss of the spatial functionality. Groups are typically used to represent the results of geometric operations.

The representation of spatial primitives - topological simplexes - is straightforward. A point is specified through three coordinates - floating point numbers. Other simplexes are specified via their definition points. The constructor of a simplex automatically checks the consistency

of its data, e.g. in the case of triangle the definition points should not lie on the same line. A simplex incorporates a corresponding representational data type. Instances of a representational data type are not independent stand-alone objects. They exist in a database only as parts of spatial objects. They disappear together with the spatial objects that incorporate them. Representational data types are used primarily to permit a more compact storage of data.

1.2.2 Spatial Functionality

The number and types of diverse geometric predicates and operations which can be applied to 3D-objects depends on the application domain. We started with the operations primarily required in 3D interactive geological modeling (Balovnev *et al* 1997) trying to keep the interface as general as possible to permit reuse in other domains. Currently *GeoToolKit* supports the following operations: intersection of spatial objects, partitioning of a space/spatial object by a plane, clipping a part of a space/spatial object by means of bounding box, equality and containment checks.

GeoToolKit distinguishes between geometric predicates returning *true* or *false* (*equal, intersect*, *contains*) and geometric operations (*intersection, division*) returning a new spatial object. Geometric operations are algebraically closed: the result of a geometric operation is a new spatial object, usually a group which can be stored in the database or used in other geometric operations. Geometric operations can produce an object which is not contained in the original class hierarchy (e.g. an intersection of two triangles can result in a quadrangle). In this case the resulting object is automatically decomposed (depending on the dimension) either into a triangle or a tetrahedron network. An operation applied to a complex object often results in a set of separate disconnected objects which can not be transformed into a single complex: in this case the result is represented as a group of objects.

1.2.3 Spatial Retrieval

Spatial objects are maintained within special containers - *spaces* - capable of the efficient retrieval of elements according to their location which can be specified either by a point or by a spatial interval - bounding box. A family of *retrieve* methods provides a convenient program interface for the spatial query manager. In the simplest case a retrieve member function takes a bounding box as a parameter and returns a set of spatial objects contained in or intersected by this bounding box. A user can formulate his query in the *ObjectStore* style as well.

Since all operations in the Cartesian coordinate system are considerably faster for cuboids than for other objects, the approximation of spatial objects by their bounding boxes is intensively used as effective pre-check in geometric operations and spatial retrieval. To provide an efficient retrieval a space utilizes spatial indexes. In addition to the embedded spatial indexes, the user can implement and integrate his own indexing method within *GeoToolKit*. In order to make an arbitrary user-defined index known to *GeoToolKit*, it must defined as a specialization of the abstract *AccessMethod* class.

A spatial retrieval involving "indirect" spatial predicates (e.g. intersects) is usually decomposed into two sequential steps. On the first step (pre-selection) the query manager retrieves all objects which intersect the bounding box of the given object. On the second step it checks whether the pre-selected objects really intersect the given object.

1.2.4 Representing Complex Objects

Simplexes per se are rarely used for the representation of real-world objects. They are more interesting as primitive building blocks for the representation of complex 3D-shapes. According to the simplicial complex approach (Egenhofer *et al*. 1989) an arbitrary shape can be approximated and then decomposed into a set of adjacent elementary components -

simplexes of the same dimension. The main advantage of such representation is that geometric operations on arbitrary complex shapes are reducible to the operations on the restricted set of primitives. Such representation is especially beneficial for the maintenance of arbitrary non-regular shapes which are typical for the majority of geo-applications (Breunig *et al*. 1994). As a default representation for a curve we use a polyline. A surface is approximated as a connected set of adjacent triangles, organized in a special data structure referred to as a triangle network. A solid can have two alternative representations: either a bounding surface approximated by a triangle network or a tetrahedron network. The first one, widely used in computer graphics, is more compact. However, for geo-applications a tetrahedron network has certain advantages because each internal point incorporates features characteristic to the whole solid which are always preserved after a series of diverse geometric manipulations and transformations. As default representations for the classes *Curve*, *Surface* and *Solid, GeoToolKit* uses classes *Polyline*, *TriaNet* and *TetraNet,* respectively.

As mentioned above, an arbitrary geometric operation on two complexes is reducible to the sequence of operations on pairs of simplexes. Obviously, in the case of two large-scale complexes the straightforward application of the operation to every combination of simplexes will fail. The optimization is concerned primarily with the restriction of simplexes which participate in the operation. In other words, a special organization of simplexes is required which could provide a fast pre-selection only of such pairs of simplexes which are really involved in the operation. For this purpose, some operations use the neighborhood information stored explicitly with every network element. Additionally complexes are supplied with internal spatial indexes which allow an efficient pre-selection of simplexes located in a given subspace. Using inheritance and function overloading mechanisms the developer can easily substitute the default optimization strategy with his own one.

Naturally, the application developer can still prefer to use a closed boundary surface to model a solid - then the built-in surface type should be employed. However, the semantic of geometric operations and predicates will be different from those expected from a filled solid.

In general *GeoToolKit* does not restrict the users to its representations. Complex spatial objects contain a reference to a dependent data structure referred to as the object representation. Abstract classes (*Surface, Solids*.) simply delegate the functionality to the current representations (*TriangleNet, TetraNet*). Such a two layer architecture allows multiple representations for the same object (e.g. one representation for compact storage, another more redundant one - for efficient computations). A complex object can change its representation without changing the object identity. This feature is of extreme importance in the *ObjectStore* context since an object can be referred to from multiple sources. User-defined representations may be used to increase the efficiency of particular geometric operations.

1.3 Developing Applications on Top of *GeoToolKit*

GeoToolKit has been successfully used for the development of several applications. The first and the most advanced one is *GeoStore* - an information system for the management of geologically defined geometries (Bode *et al.* 1994). Initially it was developed without *GeoToolKit*. Our *GeoStore* experience influenced to a large degree the architecture and functionality of *GeoToolKit*. Before launching the development of new applications, we decided to re-design *GeoStore* on top of *GeoToolKit* in order to appreciate the benefits this approach would bring (Balovnev *et al.* 1997).

The intention of *GeoStore* was to supply geo-scientists with a tool which would provide a consistent storage and efficient access for the data involved in all stages of interactive 3D-modeling using a modern database technology. The starting point for the interactive geological 3D/4D-modeling is the digitization of geological *sections* gained from open-cast workings. A section is a geological abstraction obtained as the result of an intersection of a

vertical plane with geological strata and faults. Geological strata are sheet-like structures in the earth with different mineral composition, texture and/or grain size. A fracture in the earth's materials along which the opposite sides have been displaced is known as a fault. A stratum is modeled as a list of layered bodies which are usually specified by their bounding surfaces. A fault is modeled as a simple surface. Within a section strata and faults are represented as point sets grouped in stratigraphic and fault lines. Every point in a section contains 3D-coordinates complemented with geologically-specific data such as stratigraphy. The second step in the interactive 3D-modeling is the generation of *triangulated surfaces* spread between sections. The final step which is a part of our current work, is the transition from stratigraphical bounding surfaces to *volumes*.

The main design principle we followed was to inherit from the geometric kernel as much functionality as possible. For every geological entity we found its geometric "entity" which served as its ancestor in the class hierarchy. For example, a geological stratum is represented now as a specialization of the geometric entity *Solid* which was extended with geology-specific features (e.g. stratigraphy) and relationships (e.g. the *Hanging* stratum). By using inheritance, we got a representation in which a geological entity could be treated as a geometric object. *GeoToolKit*'s geometric kernel was able to operate directly on the collections of geological objects without intermediate transformations. Topological relationships (e.g. neighborhood) provided very useful "hints" not available earlier in the "pure" geometric world. In order to benefit from this additional information we simply re-implemented virtual functions. An overridden function either substituted the geometric function completely or (more often) performed a kind of pre-selection for the geometric function calling it with the restricted subset of spatial objects.

Another application built on top of *GeoToolKit* and which is worth mentioning in the current context is *GeoDeform* which performs advanced geological modeling and animation (Alms *et al.* 1997). The *GeoToolKit*-based database maintains time varying geological strata and faults. They have a static geometry as defined in *GeoStore* with special dynamic extensions to deal with time. Direct access to the database from *GeoDeform* was not possible because of both hardware and software incompatibilities. We had to organize their interoperation using standard UNIX-sockets and a remote procedure call mechanism.

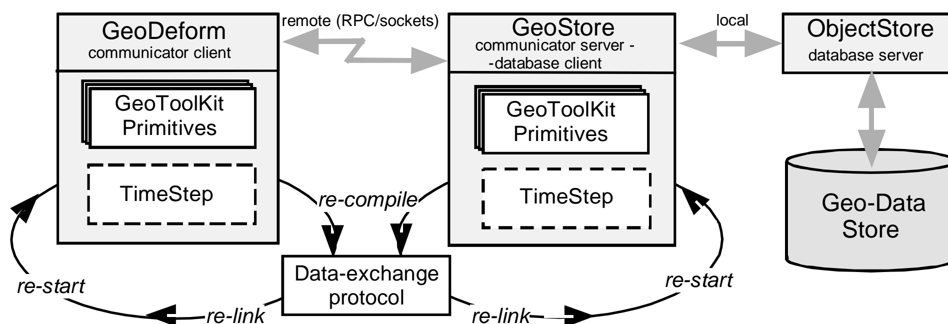

Fig. 2 Remote access to the GeoStore database in GeoDeform

1.4 Remote Access To *GeoToolKit*-based Data Stores

Common underlying data structures provide a necessary basis for the successful integration and interoperation of geo-scientific systems built on top of *GeoToolKit*. Shared object-oriented geodata stores can serve as mediators for data and services exchange. However, as mentioned above, a direct database-level client-server communication with *GeoToolKit*-based data is not always possible because of the extreme heterogeneity of the already existing applications and hardware platforms we have to deal with. Many applications simply are not available for the modifications required for the direct integration with *ObjectStore*. Database-

level communication facilities supported within DBMSs are not usually flexible enough for the advanced navigation in the network. Apart from this in *ObjectStore* all data processing (including queries) is carried out at the client site. In some cases this can lead to an unjustifiable overload of the network. For example, when we need to pick a single object from a large collection, the whole collection has to be transmitted through the network from the server to the client in order to test the selection predicate. The alternative solution is to perform the selection on the server site and transfer to the client only the selected object. That means that instead of transferring raw data through the network it is preferable to ship an operation which would be applied to the corresponding data on the server site.

Standard UNIX-sockets and RPC-mechanism turned out to be very efficient for the direct communication between two particular applications. However, these services are optimized for performance, rather than ease of programming, reliability, portability, flexibility and extensibility. However, the most serious limitation is the lack of generality. Communication conventions are hidden from the other world. Another application cannot be made aware of these internal conventions other than after a painful re-design. To avoid this we need a kind of standard distributed object computing platform. Taking into account the object-oriented nature of data the most suitable solution is *Object Management Architecture* (Object Management Group 1997) which promises to become a world-wide standard. Using this approach any other CORBA-compliant application can get an open access to *GeoToolKit*-based data stores.

The advantage of CORBA is that it delegates much of the tedious and error-prone complexity associated with low-level socket-layer programming to its reusable infrastructure. An application needs only hold a reference to a target object. The ORB is responsible for automating other common activities which usually include locating a suitable target object, activating it if necessary, delivering a request to it, and returning any response back to the caller. Developers can focus on more essential things than analyzing diverse networking troubles. Parameters passed as part of the request are automatically and transparently marshaled by the ORB, that ensures a correct interoperation between objects residing on different platforms.

CORBA object interfaces are described using an Interface Definition Language (IDL). Since the IDL specifications are automatically translated into the programming languages potential inconsistencies between the client stubs and the server skeletons are significantly reduced, providing a higher degree of type safety. For large distributed systems the loss of micro-level efficiency inevitable for such universal systems is compensated by the increased extensibility, robustness, maintainability, i.e. macro-level efficiency.

CORBA defines a flexible distribution model for transient objects. However, we have to deal primarily with persistent objects, maintained within object-oriented DBMSs. These facilities are still not standardized. To capture persistency we used a method based on the substitution of *the Basic Object Adapter* (BOA) by a specialized *Object Database Adapter* (ODA), which extends BOA by extra functions for persistent objects management and provides a tight integration between the ORB and persistent object implementations. We chose *Orbix*® as an implementation platform because it already provided a special *ObjectStore* ODA which is completely responsible for the management of persistent objects within *ObjectStore*.

The fact that we have to deal with large data stores which are permanently in use by multiple geo-scientific applications imposes additional requirements on the interoperation architecture. First of all, making these data stores CORBA-compliant should not disturb already existing applications. Since a schema evolution may be a very exhausting process for large data volumes, in ideal case an arbitrary data store should be made CORBA-compliant without changing where and how the data have been already stored. To achieve this, we just used the database wrapping technique. A wrapper encapsulates the underlying data and mediates

between data stores and diverse geo-scientific applications. In our case the wrapper encapsulates access to all *GeoToolKit* classes. While reproducing the *GeoToolKit* class hierarchy in CORBA's IDL we tried to keep the interfaces as close as possible to the original C++ ones (Fig. 3).

```
exception GTK_ObjectExists {};
exception GTK_ObjectNotFound {};
    ...
interface GTK_Space : GTK_Object {
// operations on spatial objects
void insert (in GTK_SpatialObject
    raises (reject);
void remove (in GTK_SpatialObject
    raises (GTK_ObjectNotFound);
GTK_SpatialObject
    retrieve (in GTK_BoundBox bb);
GTK_Space
    intersect (in GTK_SpatialObject
    ...
};
```

```
interface GTK_SpatialObject : GTK_Object
// spatial predicates
boolean intersect (in GTK_SpatialObject
    ...
// spatial operations
GTK_BoundBox getBoundBox ();
GTK_SpatialObject
    intersection (in GTK_SpatialObject
};
interface GTK_Line : GTK_SpatialObject
// functions specific for the lines
    ...
};
```

Fig. 3 Fragments of IDL specifications for *GeoToolKit* classes.

Within the TIE binding technique a skeleton generated by the IDL compiler corresponds to a TIE object. According to the proposed approach, every TIE object class gets a reference to a *mediator* class which in turn contains a reference to the real database object. The goal of the mediator is to provide a correct mapping between CORBA-compliant objects and database objects. When a mediator class is activated by a TIE object it converts the parameters and forwards the function call to the corresponding database object. When the operation is completed, the mediator converts the result back to the CORBA-compliant representation and returns control to the TIE object. This enables the reuse of the entire functionality of *GeoToolKit* with minimal expenses. A request for the creation of a new object at the client site results in the creation at the server site of the triple of objects: TIE object, mediator object and database object.

Spatial objects are usually not maintained as separate database entities with external names. They are typically encapsulated in large collections - spaces which usually serve as entry points in spatial databases. A particular spatial object is accessed indirectly via diverse retrieval functions associated with the space class. Our technique is based on the complete delegation of the spatial functionality to the *GeoToolKit class* library. That means that on completing retrieval we have at the server site a set of database objects which satisfy the selection predicate. However, for the communication with the client we need not database but TIE objects. The straightforward method is to extend every database object with the additional reference to its TIE counterpart. However, for the exiting databases this will require the modification of the database schema which is not desirable.

To avoid an exhaustive schema evolution we introduced a persistent dictionary with a single key - the database object identifier (OID). When a TIE object is created it is automatically inserted in this dictionary (Fig. 4). Having obtained a database object as a result of a spatial query the wrapper retrieves the required TIE object in the dictionary and forwards it to the client. Since OIDs are unique, the retrieval is very efficient. If a TIE- object with the specified key is not found in the dictionary (this may happen when a database object is created by a native application which communicates with the database bypassing the wrapper), the wrapper creates a TIE-object together with the related mediator object and inserts in the dictionary. A lazy strategy of dictionary filling encourages a concurrent operation on data for both native and CORBA-compliant applications. Since the native database stays unchanged all local

applications can continue to work with the data store as before. Moreover, any already existing data store can be made available for CORBA-compliant applications at any time. Due to the well-defined separation of database and communication objects the CORBA-compliance can be easily eliminated when not required.

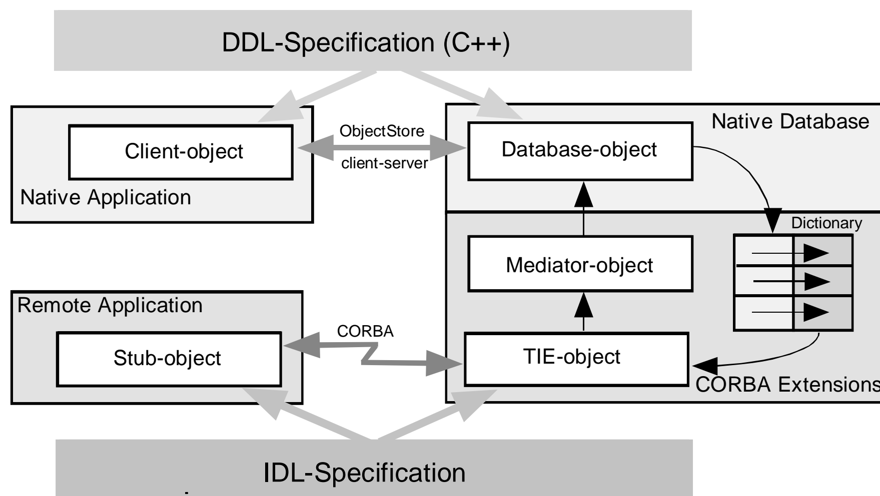

Fig. 4 Wrapper architecture

1.5 Conclusions and outlook

Our experience with the development of various applications on top of *GeoToolKit* demonstrated that due to *GeoToolKit* an application developer could focus on the application semantics instead of such "creative" tasks as optimal assembly of the spatial objects from multiple relational tables or the implementation of routine geometric algorithms. This results in a considerable reduction in the code written and makes the sources more understandable. Data and functions which were previously hidden in a particular application become available for all other applications built on top of *GeoToolKit*. Common underlying spatial data types encourage a consistent and non-redundant maintenance of data within the framework of *GeoToolKit*-based applications. Using IDL specifications of the *GeoToolKit* classes, any CORBA-compliant remote application can get the access to services associated with a database object. However, the introduction of a conventional „integrated“ object model is still a task of the highest priority. Without a standard model opening of application-specific data stores can not be considered as fully accomplished: every new external applications will still need to be tuned to the *GeoToolKit* data model.

In a recently started project, *GeoToolKit* is intended to be used as a platform for an open distributed environment that will provide an open access to the *GeoStore* database for various remote geo-services involving the GOCAD®-based 3D-modeling application located in Bonn and the geophysical modeling tool IGMAS® residing in Berlin. A free data exchange via a common database gives geo-scientists an opportunity to perform a cooperative adjustment of geophysical density and geological stratigraphic models.

Acknowledgments

This work was done in the close cooperation with the Geological Institute at the University of Bonn. We thank Prof. A. Siehl, R. Alms and T. Jentzsch who provided necessary geological backgrounds for the work. Special thanks to Norbert Klein, Dirk Moebius and Wolfgang Mueller who made a significant contribution in the implementation of software.

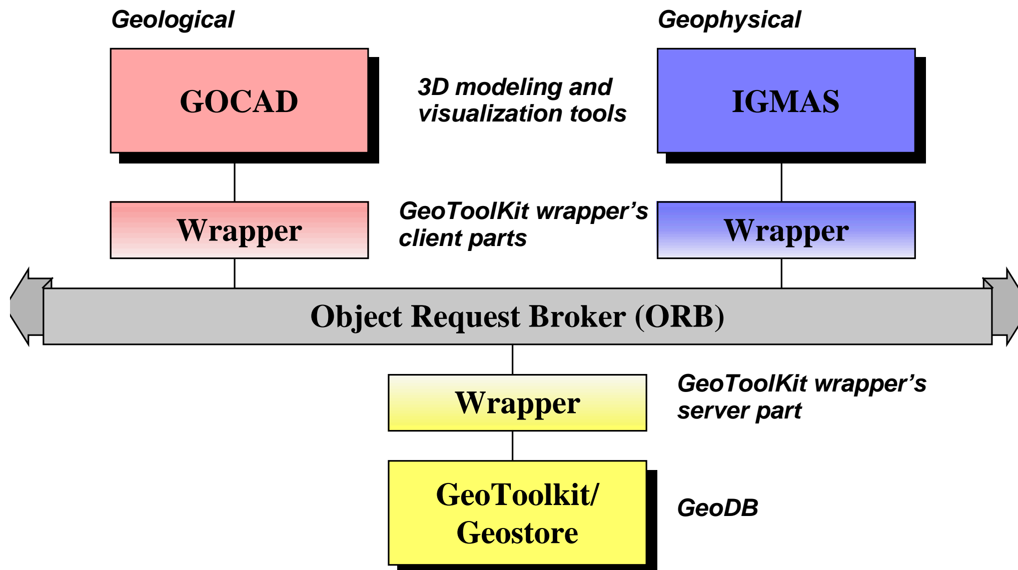

Fig. 5 Integrated environment for 3D applications

References

Alms R, Balovnev O, Breunig M, Cremers A B, Jentzsch T, Siehl A 1997 Space-Time Modeling of the Lower-Rhine Basin Supported by an Object-Oriented Database. *Physics and Chemistry of the Earth*.

Balovnev O, Breunig M, Cremers A B 1997 From GeoStore to GeoToolKit: The second step. In: Proceedings of the 5th International Symposium on Spatial Databases, *Lecture Notes in Computer Science* 1262, Berlin, Springer: 223-37.

Bode T, Breunig M, Cremers A B 1994 First Experiences with GeoStore, an Information System for Geologically Defined Geometries. In: Proceedings IGIS'94, *Lecture Notes in Computer Science*, 884, Berlin, Springer: 35-44

Breunig M, Bode T, Cremers A B 1994 Implementation of Elementary Geometric Database Operations for a 3D-GIS. In: Proceedings of the 6th Intern. Symposium on Spatial Data Handling, Edinburgh: 604-17

Egenhofer M J, Frank A, Jackson J 1991 A Topological Data Model for Spatial Databases. In: Proceedings SSD'89, *Lecture Notes in Computer Science* 409. Berlin, Springer: 271-85

Object Management Group. CORBA 2.0/IIOP Specification. OMG technical document formal/97-02-25. http://www.omg.org/corba/corbiiop.htm

J. Rumbaugh, Blaha J, Premerlani W, Eddy F, Lorengen W 1991 *Object-Oriented Modeling and Design*. New Jersey Prentice Hall