From Object-oriented Database Management Systems to High Productivity Software Development Environments

Duri Schmidt

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-014

June, 1991

From Object-oriented Database Management Systems to High Productivity Software Development Environments

Duri Schmidt
Oregon Graduate Institute
schmidt@cse.ogi.edu

1 Introduction

Object-oriented data management systems (OODBMS) differ from traditional database systems in several respects. The main difference is that OODBMS offer greater extendibility, the inclusion of user defined behavior, and enhanced reusability of prior work. While traditional database systems are able to cope with and manipulate a fixed set of predefined but parameterizable database types, OODBMS allow the programmer to define new abstract data types. The definition of a data type includes not only its structure but also the behavior of its instances. The definition of new data types is facilitated by inheritance which allows one to adapt previously developed data types to new requirements.

Much work has been done to develop several viable OODBMS. These projects usually concentrate on defining an object and database model and on developing a database management system for these models. Normally these systems offer some low level integration of the database management system with a programming language, in order to allow access and manipulation of the persistent objects contained in some database.

However, OODBMS, like traditional database systems, are not self-contained complete software development tools. They must be combined with other tools such as user interface management systems to produce entire applications, but much less work has been done to explore the combination of OODBMS and software development tools other than programming languages. Several different software development tools are potential candidates for an integration. Thus, the goal of this project is to show advantages and disadvantages of various potential combinations and to explore the integration of OODBMS and GACLs (GACL) in more detail.

2 Application model to be supported

Before discussing the main issues we limit the scope of applications we consider. Applications targeted by this project consist of several components:

- User Interface: An application presents itself to the user through its user interface. A user can access the services of the application by issuing commands. Depending on the user interface technology used, commands are induced by various means, as, for example, through written commands or through the manipulation of graphical dialog elements such as buttons or menus. Results of the execution of commands are shown by the user interface in a textual form or graphically. Our project considers only user interfaces based on a modern graphical user interface paradigm using direct manipulation, bitmapped graphics, multiple windows, and at least a keyboard and a mouse as input devices.

- Management of persistent data: Another component of an application is concerned with the manipulation of persistent data. In applications of interest to this project, persistent data is contained in persistent objects which are managed by an object-oriented database management system.

- Rest of the application: The last component of an application is called the rest of the application. It includes everything not covered by the user interface component and the management of persistent data. An example are the nonpersistent collection classes. This component usually also manages any access to operating system services.

Another assumption about the applications concerns the transaction model. The target applications use a simple pessimistic transaction model based on locking. Savepoints are possible but otherwise no fancy features are assumed.

Prototypical target applications include graphical editors such as a schema editor for a database system or form-based applications such as a bug reporting system.

We now consider how the development of such applications could be facilitated.

3 Integration of OODBMS and traditional software development tools

First we shall consider tools developed in the context of non-object-oriented, non-database applications and in the context of traditional database systems.

3.1 User interface toolkits

A user interface toolkit is a library of data types and procedures which implements common user interface elements such as menus, scrollbars or radio buttons. A programmer can use these data types in his or her program to construct a user interface for the application. The library usually also contains procedures to access events from input devices as mouse and keyboard. Examples of user interface toolkits are the Macintosh Toolbox [Appl85] and the Xt Toolkit for the X Window System [Youn89]. The problems with toolkits are: (1) they usually are not programmed to take advantage of object-oriented programming techniques, (2) the level of abstraction is too low, and (3) they do not provide an architecture for an application. We therefore decided against integrating OODBMS with user interface toolkits based on conventional programming paradigms.

3.2 User interface management systems based on the Seeheim architecture

A user interface management system is a system in which a user interface for a given application can be specified declaratively. It furthermore automatically manages the processing of user events as far as the user interface is concerned on the basis of the user interface specification [Olso87]. Often a graphical editor is used for the specification of user interfaces. A programmer can assemble the layout of a window from predefined display and data entry fields like text fields, radio buttons, check boxes, graphical output areas, pop-up menus, buttons and menus attached to a menubar. The user interface constructed with such a system dispatches user events to the items of a layout. The dialog items process related events on their own or call attached user defined actions. User interface management systems are usually employed in the context of applications which don't use a database system for the management of persistent data.

Many if not most of the user interface management systems are based on the Seeheim application architecture [Gree85]. In the Seeheim model the user interface is clearly separated from the other parts of the application. The model assumes that the user interface and the other parts of an application are only loosely coupled and that communication between the user interface and the other parts may be accomplished by a low bandwidth connection [Myer89]. One could indeed run the user interface in one process and the other parts of the application in anther process. The user interface is further divided into a presentation component, a dialog control component and an interface to the other parts of the application (Fig. 3.1). The dialog control component dispatches and processes user events, the presentation component is responsible for drawing the user interface, and the interface to the other parts of the application connects the user interface to the other parts of the application, i.e., to the so-called rest of the application along with the persistent data management part.

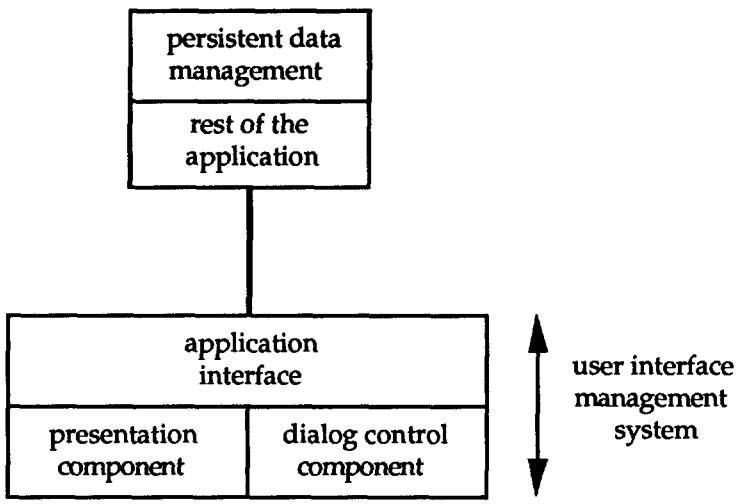

Fig. 3.1: Seeheim model

Communication between the user interface and the other parts of the application is accomplished through shared variables, message passing or a procedural interface.

One advantage of this architecture is that a new user interface can be used without altering the rest of the application as long as the interface between them is not changed. Thus the user interface can easily be modified to take advantage of new user interface paradigms or new user interface techniques without the necessity of changing the other parts of an application. Separating the user interface from the other parts of an application also facilitates the incremental development of the user interface because the other parts of the application are not affected by these nonfunctional changes.

Another advantage of this architecture is that the non user interface parts of an application are relieved of the burden of processing every single user input. The user interface of an application is thus relatively autonomous, i.e., much of the user input can be processed locally in the user interface without the necessity of accessing other parts of the application. For example, when a user fills in a form the user interface processes user inputs until the user has committed the changes. Only then is the application sent the final content of the form.

A final advantage is the declarative nature of these tools. The programmer need no longer worry about the sequence of events and can concentrate her or his efforts on other parts of the application.

Unfortunately this architecture has several shortcomings:

- A dialog can only be assembled from a predefined set of dialog items. While few systems based on a textual specification languages allow the extension of the set of dialog items, for example Apollo Computers Open Dialogue [Comp86], even fewer systems with a graphical user interface editor provide an extension feature. Even then, the definition of new dialog items is normally difficult. Thus it is either not possible or at best difficult to enhance the user interface with domain specific interaction techniques.

- Only form-based user interfaces are supported, but only a fraction of the applications which fit the application model we want to support have a form-based user interface.

- The basic assumption of the low bandwidth communication between the user interface and the rest of the application does not hold for many applications, especially graphical editors. In these applications communication between the user interface and the rest of the application is very frequent because the rest of the application must often be consulted to provide the semantically correct feedback, for example during dragging or resizing graphical objects. Some systems, such as Higgens [Huds88], try to provide this kind of feedback in the context of a user interface management system based on the Seeheim architecture. Unfortunately however, much of the declarative nature of other user interface management systems is lost. If the Higgens specific terms are stripped off the system resembles a somewhat modified model-view-controller architecture where the model is constructed of active values triggered by user or application events.

- An investigation [Ross87] also showed that many designers have difficulty separating the user interface from the other parts of the application.

- The division of an application into a user interface and other parts gives the application developer no further advice for the architecture of the other parts of the application.

- In these systems, reusability is provided through the user interface management system and not through the inheritance mechanism usually used in object-oriented systems.

- These tools do not support the development of application specific aspects not related to the user interface by high level tools.

- In the case of a textual specification a programmer has to learn a new special language.

Recently several research groups started to explore the integration of user interface generators and user interface management systems with object oriented database systems. Facekit [King89] and LOOKS [Alta89] [Plat89] combine an OODBMS and a user interface management system. They exhibit most of the previously mentioned advantages and disadvantages. Usually a nested object representation for complex objects is derived from the database schema and it is possible to call methods of the displayed persistent objects. In LOOKS the user interface is called through

global procedures from the database application thus deviating from the programming paradigm used in the non user interface related parts of an application. Neither system supports the construction of graphical editors.

Another interesting research project [Flyn90] tries to unify persistent data management and user interface specification management. User interface specifications are also considered as persistent objects. They store both a description of the display of the persistent object and rules applied to process user events. If a persistent object has to be displayed and manipulated, a specification object is activated. The specification and the persistent object are used to generate a display object, which displays the persistent data and controls the processing of user events. Every persistent object can be combined with several different specification objects. Because the system supports inheritance, specification objects can be easily reused to construct new ones. And because specification objects are persistent objects any tool provided by the database system to modify persistent objects can be used to alter the content of these objects during program execution, giving the system much flexibility.

After considering advantages and disadvantages of these various systems we decided not to investigate the integration of a user interface management system and an OODBMS although this approach could be very reasonable for applications using a form-based user interface. But we feel that it would be advantageous to retain in our approach at least some of the declarative flavor of the user interface management systems.

3.3 4th generation languages

In the context of advanced database systems based on traditional data models such as 4th Dimension from Acius [ACIU86], the high level tool support covers the interactive definition of database schemas, the specification of the layout of windows, and the formulation of queries and reports through graphical editors. Systems as these are called 4th generation languages [Olso87]. This actually is a misnomer because the languages in which the rest of the application is programmed do not differ from third generation languages except for the more or less elegant integration of a persistent data type. The user interface and the management of persistent data are normally tightly coupled in 4th generation languages. After specifying the database schema, a default user interface is often inferred from the schema and one can then enter new data and issue queries whose results are displayed through the default user interface. A programmer can adapt the default user interface to his or her own taste or can define completely different ones. Typically, dialog items such as text entry fields are bound directly to fields of a record and the system automatically propagates to the database data entered in a panel. It is usually not possible to relax this tight coupling of user interface and database or to have dialog items for data computed from persistent data or for data from which persistent data is infered.

4th generation languages have the following advantages:

- User interface design and user event processing are simplified.
- Not only the user interface but also the management of persistent data is supported through high level tools.

And the following disadvantages:

- Dialogs are assembled from a predefined, usually not extendible set of dialog items.
- Only form-based user interfaces are supported.
- The database and user interface are normally tightly coupled, i.e., it is often not possible to have special fields to display data derived from the persistent data. We think that the tight coupling is not reasonable for many applications we would like to support.
- These systems do not support the development of the application domain specific aspects of an application not related to the user interface or persistent data management through higher level tools than the programming language.
- If a program needs facilities which cannot be produced with these tools a programmer must use the low level programming language which is part of the whole system. There is no intermediate level of support between the high level tools and the low level programming language.
- 4th generation languages usually provide only a basic transaction mechanism built into the language. They don't support different transaction patterns through higher level abstractions.

The consideration of the advantages and disadvantages of 4th generation languages led us to the decision that we don't want to develop a 4th generation language using an OODBMS instead of a relational database system. But it would be interesting to have a system with similar tools supporting applications with more complex user interfaces less tightly coupled to a database.

3.4 Generic database browsers

Another less ambitious approach to combining a database and a user interface are generic database browsers such as the database browser of SunSimplify [Sun 88]. With this browser every database has an application independent user interface and every database can be examined and modified.

An example of such a system in the context of OODBMS is the KIVIEW browser [Laen89]. It provides a standard user interface for every database. It is also possible to customize the user interface. KIVIEW therefore provides functionality similar to the systems mentioned 3.2 or 3.3.

Because such a generic database browser is valuable tool, we would like our approach to provide an application independent standard user interface.

Having seen several approaches we turn now to our own approach of the integration of OODBMS and other software development tools.

4 Integration of OODBMS and GACLs

4.1 Levels of integration

Our approach is based on a model with three levels of integration between OODBMS and other worlds (Fig. 4.1).

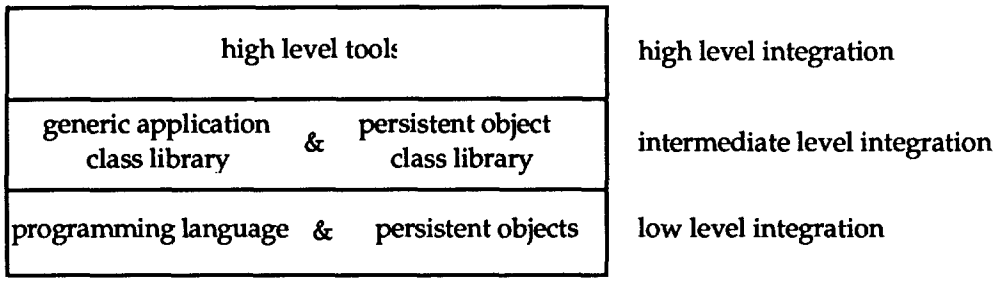

Fig. 4.1: Levels of integrations of OODBMS and other software development tools

The lowest level of integration is the integration of an object-oriented programming language and an OODBMS through persistent objects. We define an OODBMS to be a database system which is specialized to manage persistent objects [Schm90]. An object-oriented database consists of persistent objects. Persistent objects are objects having a possibly long lasting existence. This means that their existence is not confined by the process in which they are instantiated. Persistent objects differ from nonpersistent objects only with respect to persistence. This means that persistent objects also have state and behavior, and that they are instances of a class. Classes may inherit from each other and dynamic binding is supported. This level of integration is addressed by projects developing an OODBMS. It is not part of this project. Instead we will use the existing combination of the programming language C++ [Stro86] and the OODBMS DB++ [Schm90].

The next level of integration in our model, the intermediate level, is the integration of an OODBMS and a GACL that is not specifically built to be used with an OODBMS. GACLs are a new emerging software technology in the field of object-oriented software development. The classes of a GACL are organized so that they form a generic application, i. e., an application having all the common features and properties exhibited by all applications of a certain type [Schm86]. The result of this intermediate level integration will be a new GACL. The generic application of the new class library must exhibit all the common features and properties of applications that have a modern graphical user interface and which use an OODBMS for the management of the persistent data.

The last level of integration in our model is the integration of the OODBMS and high level tools, e.g., user interface generator or schema editor. These tools are not independent of the GACL. They generate new classes, which often inherit from classes in the GACL or they generate specifications which can be interpreted from instances of some classes of the library.

We expect that this three layered application architecture has the following advantages over the other mentioned approaches for integrating an OODBMS and other software development tools:

- Support of a wide range of applications which can have either a form-based, or a non form-based user interface, or a combination of both. The Garnet system [Myer90] shows that such a system can support not only form-based user interfaces but also in certain types of graphical editors.

- Graceful degradation of high level tool support. If a program needs facilities which cannot be produced with a high level tool, a programmer eventually can use existing classes from the GACL. Domain specific knowledge may be provided by these classes, or at least domain specific classes can be derived from the classes of the library.

- Easy integration of an application independent database browser. Because form-based user interfaces will also be supported adding a generic database browser will be easily possible.

- Application development cost and maintenance cost is reduced because the structure of applications is predefined and because many parts of an application can easily be derived or composed from those already defined in the GACL.

Another reason for our approach is the inadequacy of persistent object class libraries currently delivered with OODBMS. These rarely contain more types than a conventional database system provides as fields or as set types [Maie87] [Onto88]. Therefore at the beginning of using such a system they don't provide more predefined, reusable types than conventional database systems. The integration with a GACL offers the opportunity to provide at least abstract persistent object classes for the classes from which the programmer most often derives his or her own classes.

Together with a compiler and other language related tools, these tools build an open ended high productivity software development environment for applications featuring a modern graphical user interface and persistent data management by an object-oriented database management system.

We will now explain the strategy for the intermediate level integration. Then we will show the properties of GACLs and OODBMS in more detail. The discussion will also show the expected problems of the integration. Finally we will discuss previous work in this field and the detailed goals of the project.

4.2 Strategy for the intermediate level integration

It is beyond the scope of our project to develop a new GACL and a new OODBMS. Our integration strategy therefore relies on an existing GACL and on an existing OODBMS. Because neither were designed at the outset to interface with each other, we will adapt the GACL to the needs of the OODBMS. Conversely we will adapt the OODBMS to the special needs of the GACL. Thus the persistent objects enhance the functionality provided by the generic application and the class library extends the capabilities of the OODBMS through such services as dialog management.

4.3 Properties and architecture of GACLs

4.3.1 Properties of GACLs

GACLs contain, as mentioned, classes which form a generic application. A GACL makes it possible for a programmer to assemble many parts of a specific application from classes of the library or from classes derived from classes of the library. The use of GACLs reduces the complexity of the development of applications with a modern graphical user interface because dialog elements, event dispatching, and event processing are predefined, because many other building blocks, such as collection classes, are provided, and because they can easily be adapted to special needs. The use of GACLs also facilitates the enforcement a homogeneous user interface because the user interface guidelines are built into the dialog management part of the class library. Furthermore, the GACL defines to a large extent the structure of an application. This eases the design and the maintenance of applications.

ET++ [Wein89] is a typical example of a GACL. It has the following properties:

- The generic application standardizes the structure of an application and the general flow of control.
- It is possible to work concurrently with multiple documents.
- A document may be displayed in one or more windows.
- A programmer need provide nothing so that windows can be manipulated, i.e., moved, resized or closed.
- Windows can be divided into arbitrary clipping regions. These clipping regions show parts of virtual drawing surfaces called views.
- Clipping regions support changing the visible part of a drawing surface which is larger than the clipping region (scrolling).
- If graphical objects are moved over a drawing surface, the clipping region automatically adjusts the visible part of the drawing surface as the object is moved. Thus, the part of the drawing surface over which the graphical object is moved is always visible (autoscrolling).
- Redrawing windows is flicker-free because double buffering is used.
- Printing of documents is automatically supported.
- Many different collection classes support the programmer managing groups of nonpersistent objects.
- ET++ provides a framework to support undoable commands.

- The declarative construction of dialogs is supported through many predefined dialog elements and an automatic dialog layout feature.
- Classes are provided to manipulate multiattribute text with proportional fonts.
- ET++ contains inspectors for looking at the content of objects during runtime and a browser for looking at the definition and implementation of the classes of an application.

4.3.3 The architecture of GACLs

As before we will concentrate on the architecture of ET++ because its architecture is quite typical of similar class libraries.

The classes of ET++ can be divided into several groups (Fig. 4.2). The general base classes group contains the most important abstract classes of the ET++ class hierarchy. To this group belong the root of the ET++ class hierarchy, classes to process user events, the base class of graphical object classes and collection classes. The application framework classes realize the generic application, i.e., they constitute a specialized framework of classes, the application framework. This framework provides the general event processing structure of an application with a graphical user interface and as input devices, a mouse and a keyboard.

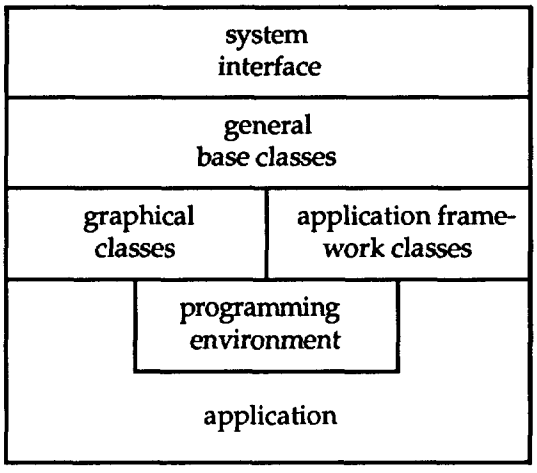

Fig. 4.2: Groups of classes of ET++

The graphical classes contain user interface items such as menus, dialog elements, scrollbars, etc., and a class to compose such items. The system interface classes implement the interface to the operating system and the interface to the window system. The programming environment contains classes to inspect the state of objects and the definition of their classes at runtime.

From the general base classes the classes Object and VObject are especially important for our purpose. The class Object is the root of the ET++ class hierarchy. It defines the abstract methods to compare objects, it implements change propagation, and it implements dynamic type checking. Its most important function is its role as root class. Nearly all other classes of ET++ are derived from Object. This allows the construction of collection classes which can have any instances of subclasses of Object as element and every object can take part in change propagation.

The class VObject is the base class of all graphical object classes. It is derived from the class EvtHandler, which defines an abstract protocol to process user events. The class VObject specifies instance variables and methods to manage the location and the extend of a graphical object and to draw itself. Application specific graphical object classes usually are derived from this class.

The application framework classes define the general structure of an application which processes user events. The most important concepts implemented by the application framework classes are the concept of an application and the concept of a document. The application manages the documents, i.e., it knows how to create new documents and how to open existing documents. An application can handle many documents of the same or different types.

A document handles a closed group of persistent objects that is edited together and stored together in a file. In addition, a document manages the window, views, etc., which are used to display and manipulate the data of a document. A document also takes part in processing user events. Documents do not overlap, i.e., documents do not contain references to objects of other documents and they are not nested, i.e. a document cannot be element of another document. A document itself is also a persistent object. Its persistence is refleceted not only in some persistent data but also in the existence of a file with the name of the document. The document concept supports only one activation/deactivation policy, i.e. all elements are activated when a document is opened and elements are deactivated when a document is closed. It also provides for only one clustering policy, i.e. all elements are clustered in one file. A document can access its elements only by navigation because no query facilities are available.

Documents also represent transactions. Every document is an independent transaction because documents don't overlap. The open, revert, save and close commands of the application framework correspond to start transaction, rollback transaction, commit and start transaction, and commit transaction respectively. Running multiple transactions in parallel in one process works because by convention documents don't share objects and because GACLs provide uniqueness of object identifiers only within one document even though there is no system enforced isolation between the transactions. In the context of an OODBMS, however, identifier usually are unique within a database and multiple documents may be stored in one database. Current OODBMS also allow an application to run only one transaction at a time.

Transactions are linked strongly with the opening respectively closing of documents and with the activation respectively deactivation the elements of a document. The transaction mechanism is degenerated in so far that the transactions are not related to synchronization of multiple users. The start of a transaction, i.e. the opening of a document, doesn't lock the file of the document to prevent other users from accessing the document.

The application concept is represented in the class Application and the Document in the class Document. These two classes are particularly important for the integration with OODBMS.

Differences to other generic application class libraries

Other GACLs such as MacApp [Schm86], Interviews [Lint89] or Falcon [] provide either similar or some aspects of the functionality described.

These GACLs differ not only in their functionality but also in their architecture. One main difference is whether the design of the class library is based on a single root or a forest approach. In the single rooted approach almost all of the classes are derived form a single base class. Examples for this architecture are ET++ and MacApp. The single rooted approach results in a very homogeneous class library. All classes have the same basic protocol such as change propagation and all instances of derived classes are substitutable on the level of the base class which makes combination and reuse of different objects very easy.

In the forest approach several root classes exist, one for example for the collection classes, one for user interface elements and another for the graphical objects. Promoters of this approach argue that in this approach "inheritance is only used where it is really needed for getting the required behavior" and that "this architecture is better suited to building a framework that includes contributions from a relatively large and diverse group of developers" . Another point often cited is that in the single rooted approach every object is tagged by the amount of memory needed for the instance variables of the base class even though it doesn't need these instance variables. However, this problem can be reduced sharply by an approach in which most classes are derived from the base class except for certain types which are more treated as values as for example integer, string, point, or rectangle [Wein89].

Of note also is the fact that most of these class libraries provide a group of collection classes which are heavily used elsewhere in the class library.

In our project we will discuss integration issues for both approaches where relevant. But for our prototypes we want to use the GACL ET++. We chose ET++ for principal and pragmatical reasons. It offers more functionality than comparable class libraries such as MacApp [Schm86] or Interviews [Lint89] and its architecture is more homogeneous. Its source code is available and we have experience using it.

4.4 Properties of OODBMS

OODBMS are, as mentioned, database systems specialized to manage persistent objects. OODBMS provide an extensible type system in which the structure and the behavior of new persistent objects can be defined. To reuse existing types new classes can be derived from existing ones using inheritance. OODBMS also facilitate the homogeneous usage of databases, because the persistent objects of a database can only be accessed through the methods defined in their classes.

DB++ [Schm90] is an OODBMS which manages persistent C++ objects. Persistent objects in DB++ are based on a framework approach. From several abstract classes new classes must be derived to get a database and persistent object classes (Fig. 4.3). Therefore persistence in DB++ is provided through inheritance, i.e., classes have persistent instances if they inherit from the base class of all persistent object classes DBObject. Persistent objects have strong identity and keep their persistent data in fields. A programmer can assemble new persistent object classes from a set of predefined field classes, the subclasses of DBField. But he or she can also define new field classes for special applications. A persistent object class is always defined together with a metaclass, for example DBObject together with DBObjectClass. The metaclass knows the structure of the persistent objects and an instance of a metaclass (a metaobject) represents the extension of the corresponding persistent object class. The database system as a whole is represented through a

database system class which must be derived directly or indirectly from the base class of all database system classes DBase. The database system class defines, from which persistent object classes a database may contain instances. The DB++ class library provides field classes for the most common values such as integer or small strings and persistent collection classes to cope with groups of objects such as lists.

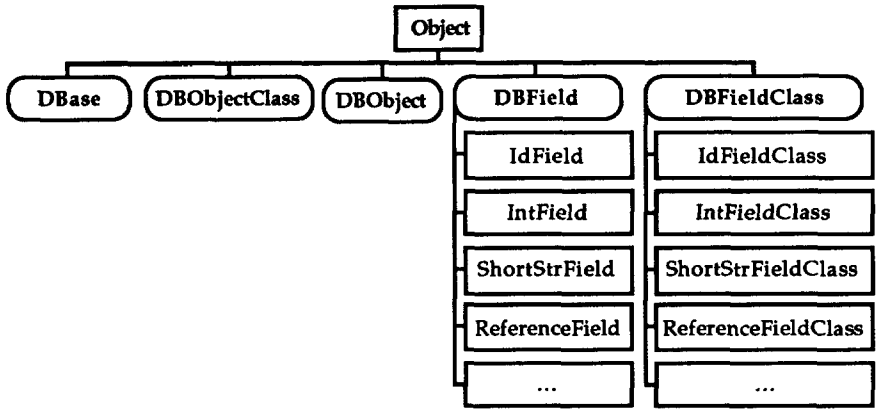

Fig. 4.3: Base classes of DB++

Beyond persistent objects OODBMS usually also provide a general transaction mechanism to synchronize concurrent access to a database; they come with a set of collection classes; they often offer query facilities and support versions of objects.

Types of OODBMS

There are three major principles used to determine whether an object is persistent or not. The first approach is based on the reachability of objects from one or more (persistent) root objects. All objects are persistent at the end of a transaction which are reachable from a root object. One therefore has to watch carefully that no objects, which should not be persistent, are referenced from persistent objects at the end of a transaction. In this project database systems relying on persistence by reachability aren't considered any further because the majority of systems which support the programming language C++ don't use persistence by reachability.

In the second approach persistence is based on inheritance. In such a system an object is potentially persistent if its class is derived from a specific base class.

The third principle to determine persistence of objects is called persistence by allocation. In this approach objects are persistent if they are allocated in a persistent memory region. With this approach persistence and type are orthogonal, i.e., every type can have persistent instances.

Overall, OODBMS are currently delivered with a rather small and poor persistent object class library. At the beginning of usage they don't provide more predefined reusable types than conventional database systems, because the systems are designed to be as generally usable as possible, and therefore nearly nothing is known about the classes which are used in the applications. The integration with a GACL improves this situation. It is at least known from

which classes a programmer usually derives and assembles application specific persistent objects. It is therefore an important goal of this project to construct a more meaningful and more comprehensive persistent object class library.

4.5 Issues in the integration of GACLs with OODBMS

There are several different issues for the integration. Some problems are caused by the architecture of application frameworks and the properties of OODBMS. Other issues of the integration have their roots in the desire to enhance and improve the functionality of the OODBMS. Another interesting aspect is the attempt to use a priori knowledge about the interaction with the database system.

4.5.1 Problems resulting from the architecture and the design of the GACL

For GACLs and OODBMS to work together one has to overcome several obstacles resulting from the architecture and the design of the GACL and the properties of OODBMS.

One problem caused by the architecture and the design of the GACL is that the current GACLs use files to manage persistent data. The structure of the application framework classes is therefore biased to use files.

Another issue is the concept of a document in these GACLs. The concept of a document as discussed above is too restrictive in the context of database systems.

A further very important problem is that current GACLs incorporate a transaction concept which only partly fits the transaction concept of OODBMS.

4.5.4 Problems resulting from the properties of the OODBMS

The problems resulting from the properties of OODBMS can be classified into problems common to all systems and issues depending on the principle on which persistence of objects is determined.

Problems common to all OODBMS

Common to all OODBMS is the problem that classes provided by the OODBMS usually are not based on the same root class as the GACL. Therefore classes of the OODBMS don't support the basic protocol of GACL classes such as change propagation, identity check, or comparison. Furthermore, instances of predefinded OODBMS classes and GACL classes cannot be mixed in collections because collections are defined on the basis of the GACL's or the OODBMS' root class or, in the case of parameterized types, on the basis of some common ancestor. This reduces the homogeneity, substitutability and interoperability of objects. This problem especially concerns single rooted GACLs and OODBMS which provide persistence by inheritance.

However, OODBMS in which persistence is orthogonal to type also suffer to a certain degree from the same problem. Often their collection classes are provided as parameterized types. The generated collection classes are of course just as little derived from the root class of the GACL as the ones provided by OODBMS based on persistence by inheritance.

Another issue concerns collection classes. Usually both the GACL and the OODBMS provide collection classes. These classes usually have different functionality, different protocols, and they are derived from different base classes. They are therefore not substitutable, and conversion operators aren't available. Collection classes of GACL's normally are targeted to cope with small to medium numbers of members whereas collection classes of OODBMS are designed to cope with very large numbers of members. Collection classes of OODBMS usually provide a much richer query facility and the capability of building indexes.

A further issue is object faulting. If a reference to a passive object is dereferenced the corresponding object is automatically activated in an OODBMS. The so called object faulting relieves the programmer from worrying about object activation. But often such objects have to be connected to nonpersistent objects in order to function properly. An example is a persistent object which consumes events. To get and distribute events the object has to be installed in the chain of event handler objects. This means that one has to be very careful about not faulting objects inadvertently, because this could lead to objects not being properly installed in their environment.

A last hindrance for the integration is that GACLs usually rely on variable length objects where some OODBMS only provide fixed length persistent objects.

Issues depending on the principle applied to determine persistence of objects

In the case of OODBMS which are based on persistence by inheritance the integration has to cope with the following obstacles:

- Some of the already defined classes in the GACL, as for example the text classes or graphical object classes, should also have persistent versions.

- Another problem concerns GACLs and OODBMS which are based on single inheritance. There inheritance is used to get the persistence property. But often it is necessary to construct classes which should have persistent instances, but which also should inherit from another class. An example is a persistent graphical object class which should inherit from the base class of the persistent object classes and from the base class of the graphical object classes.

A consequence of the orthogonality of type and persistence is that all classes may have persistent instances. The instantiation of one object often leads to a cascade of creations of other objects, i.e., in the constructor of an object other objects are instantiated. Because every object potentially can be created by such a cascading instantiation all constructors of all the classes of the GACL must be changed so that all objects created during such a cascading instantiation are allocated in the correct part of the memory. Otherwise it may happen that some parts of an object are allocated in persistent memory while other parts are erroneously allocated in nonpersistent memory. This is a tedious work since, for example, the GACL ET++ has several hundred constructors.

4.5.5 Enhancing and improving the functionality of the database system

The issues considered so far make a GACL and a OODBMS function together and relaxes constraints resulting from the file orientation of GACLs, i.e. applications can now be developed which use an OODBMS for the management of persistent data. The next step of the integration is to exploit the services offered by the GACL to enhance and improve the functionality of the OODBMS and to overcome limitations of current OODBMS.

4.5.6 Using a priori knowledge to increase performance of the interaction with the dbms

The last issue in the integration of an OODBMS and a GACL concerns the use of a priori knowledge to increase the performance of the interaction with the OODBMS. Database systems are constructed with the goal of providing persistent data management to a wide variety of applications. Therefore little can be presumed about the properties of applications and one cannot take advantage of such knowledge to increase the performance of the dbms. In a GACL, however, much more is known about the properties, the sequence and the probability of interactions with the DBMS in general and in the near future. The GACL therefore can provide the OODBMS hints so that the OODBMS can optimize its operation. Moreover, in the context of a GACL it is possible to provide an application programmer with higher level abstractions which represent typical interaction patterns with the dbms and therefore even more specific information can be provided to the OODBMS.

5 Related work

5.1 Gemstone and Smalltalk

In one project [Purd87] the OODBMS GemStone and the Smalltalk programming environment were combined. GemStone and Smalltalk run as different processes. Agent objects represent persistent GemStone objects in the Smalltalk application. The agent objects forward messages to the persistent objects in GemStone. Agents can also have a certain local autonomy, i.e., they can cache part of the state of the GemStone objects and can have their own methods. This form of integration belongs to lowest layer in our integration model. To our knowledge the integration stopped at this level of integration and no further special GemStone classes were built to ease the construction of new Smalltalk applications that use GemStone.

5.2 ETDB

Another approach is our own previous work ETDB that integrates the OODBMS DB++ and GACL ET++ [Schm90]. It had narrower goals than our current approach. In our previous work

- only the same document abstraction as currently incorporated in ET++ was supported,
- the integration of a transaction concept adapted to the needs of a OODBMS was missing,
- the integration strategy was different, and
- only one persistent object could be displayed concurrently with the standard dialog.

Several properties of DB++ facilitated its integration with ET++:

- The base class of all persistent object classes DBObject is derived from the root class of ET++ Object. Therefore a persistent object can be used at any place where an instance of the root class can be used. An example is a selection list in a graphical editor: to manage a group of selected objects which by nature constitute a temporary group, a nonpersistent list can be used.

- The protocol of the persistent collection classes is almost identical to the protocol of the nonpersistent collection classes. A programmer knowing the protocol of the collection classes of ET++ is therefore not required to learn a new protocol for persistent collection classes.

To keep the complexity low ET++ wasn't altered. A new layer called ETDB was built below ET++ and DB++. It integrates the GACL ET++ and the OODBMS DB++ (Fig. 5.1).

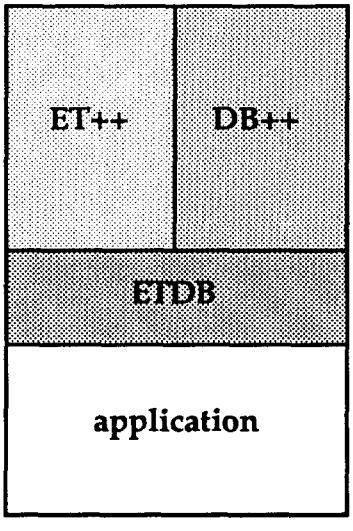

Fig. 5.1: Integration strategy

In this new layer several ET++ and DB++ classes were specialized and adapted to the needs of the integration, so for example the application class, the document class, and the database class. A persistent graphical object class was also implemented.

This effort resulted in a new GACL with which applications with a direct manipulation user interface based on the cited document metaphor can be constructed (Fig. 5.2).

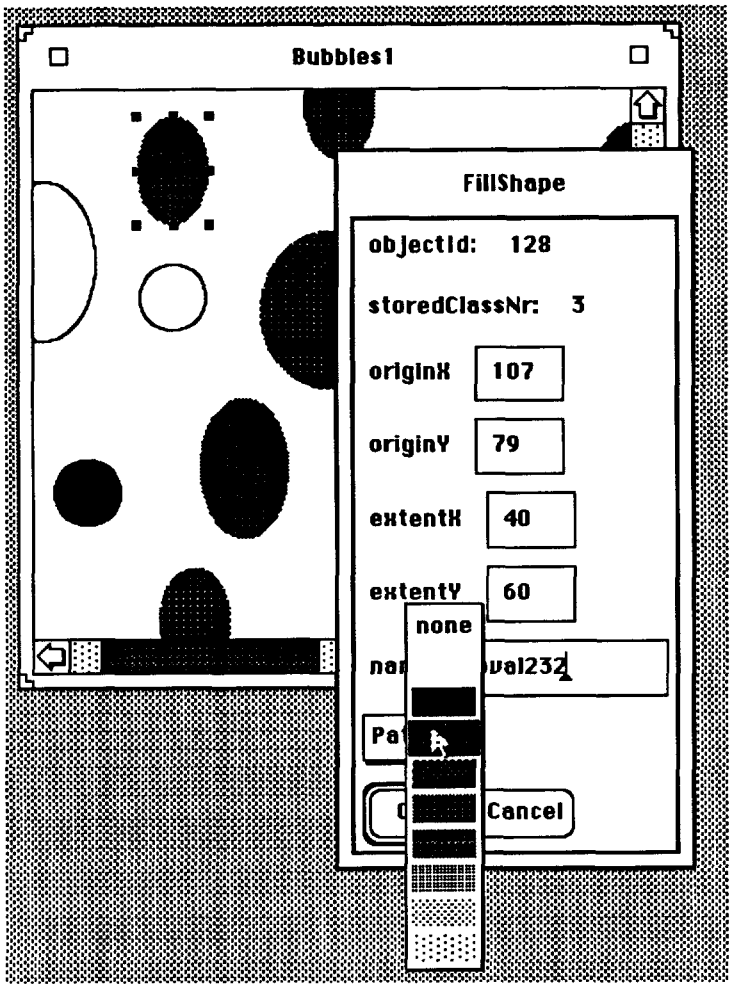

Fig. 5.2: Example application bubbles.

To increase the functionality of the OODBMS DB++, more functionality was added to the classes of the persistent object class library of DB++. Each persistent object class derived from the persistent object classes of the ETDB layer had a standard dialog. The standard dialog displayed the content of the fields of a persistent object and the values could be modified (Fig. 5.2).

The result of the integration was, as described above, the new class library ETDB. To construct a new application a programmer usually no longer derived classes from ET++ or from DB++ but from ETDB.

The ETDB approach had several shortcomings:

- The integration resulted in a strange distribution of functionality. The basic ET++ layer was file oriented. In ETDB the file dependencies were removed and replaced by database facilities.

- The basic concept of a document was not changed and the transaction mechanism was not adapted to the needs of a database management system.
- Only the display services of the GACL were used to enhance the functionality of the OODBMS.
- The standard dialog was restricted to display only one persistent object at a time.

6 ETDB++

Having explained the problems and the limitations of recent approaches, a new approach – ETDB++ – is presented and its solutions to the various problems are discussed. The solution to the various problems and issues suggest a variety of means ranging from programming guidelines, new or adapted classes to requirements for the OODBMS.

6.1 Solutions to issues resulting from the architecture of the GACL and properties of OODBMS

6.1.1 Solutions to problems resulting from the design of the GACL

File oriented application framework

The solution to the problem of file oriented application framework in ETDB++ is to build a neutral application framework which can be specialized for different kinds of persistent data management.

If we look at ETDB, we see that the integration resulted in a strange distribution of functionality. In the ET++ layer the application framework uses files, in the ETDB layer the file dependent classes are replaced by classes which convert the GACL into one prepared to use an OODBMS. The ETDB++-architecture uses an application framework neutral to persistent data management and neutral to the supported document metaphor. From the neutral application framework specialized application frameworks using different means for persistent data management are derived. This architecture results in a class hierarchy for applications as shown in Fig. 6.1. A generic application class provides the abstract protocol for applications. Its subclasses are specialized application classes representing applications with different approaches to persistent data management. The document classes will be organized in a similar manner.

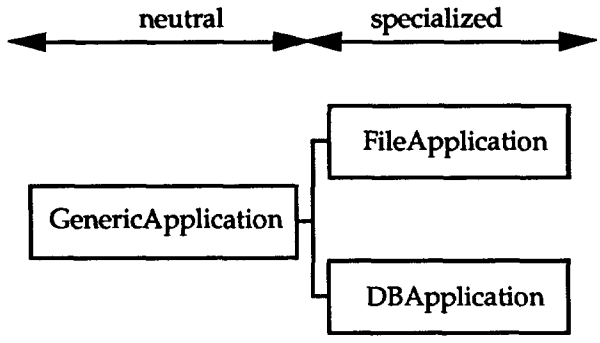

Fig. 6.1: Inheritance structure for the application classes of ETDB++

Transaction concept

Because OODBMS don't allow one process to execute more than one transaction at a time and to prevent problems resulting from the concurrent execution of multiple transactions in one process, ETDB++ allows a process to execute only one transaction at a time. A user still can run multiple transactions in parallel against one database, because he or she can start multiple application processes. Any number of documents can be opened and closed during one transaction, but they do not represent separate transactions. ETDB++ applications therefore could be described as one transaction – multiple document applications.

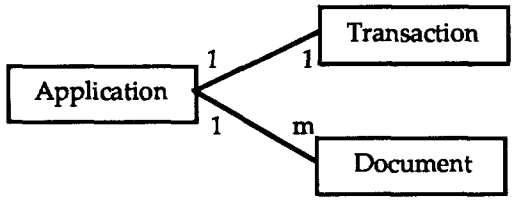

Fig. 6.2: Structure of the transaction abstraction for overlapping documents

Furthermore, ETDB++ uses the concept of master and slave applications so that users are not bothered with starting all transactions from the shell. The master application lets the user select a transaction to be carried out. It then either starts a slave application in another process or activates an inactive slave application. The slave application receives all needed information, i.e., which transaction to run against which database from the master application. At the end of the transaction the slave application notifies the master application that it is ready to work on a new transaction.

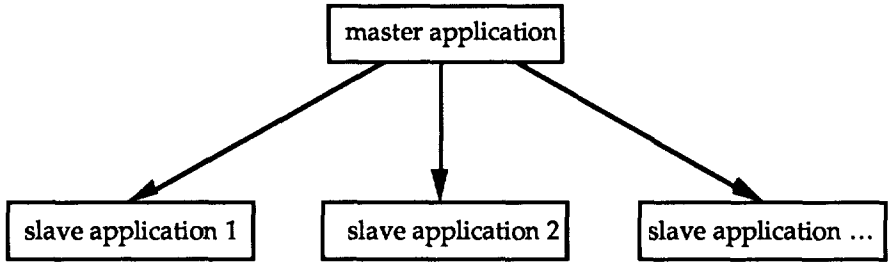

Fig. 6.3: Structure of the transaction abstraction for non overlapping documents

In ETDB++ the strong linkage of transactions, opening/closing documents and activation/deactivation of objects is broken up. A transaction can be started before a document is opened, but the application automatically starts a transaction when a document is opened and no transaction is already running. It is the duty of a document to decide about the appropriate policy to lock and activate its elements, i.e., whether to activate all objects when the document is opened or later on demand or according to another policy. Likewise it is also the responsibility of a document to implement the appropriate unlocking and deactivation policy. Other consequences of the separation of transactions from documents is that documents can be closed without ending a transaction and they can implement a deactivation policy adapted to their special purpose.

The end of a transaction causes the closing of all documents, deactivation of all persistent objects and release of their locks. This policy can be overriden and adapted to special needs of the application.

Finally one has to address the question of who manages transactions. In ETDB++ the application classes are responsible for managing transactions. Therefore they provide the necessary protocol to start, to rollback and to commit transactions. A transaction is of course an object too. Start, rollback and commit are methods of the transaction class and these operations can be used to trigger application specific actions.

Concept of a document in ETDB++

In ETDB++ a document still manages a group of persistent objects, but, as explained, a document now is only loosely coupled with transactions. Documents may differ in the following properties:

- Management of persistent data: for the management of the persistent data either files or a OODBMS is used. (file / OODBMS)
- Concurrent activation: some document abstractions assume that only one document is concurrently active in an application. Others may allow the concurrent activation of more than one document. (single / multiple concurrently active documents)
- Structure: some documents presume that the objects they manage are not shared with other documents. Others support the overlapping of documents. (nonshared / shared)
- Element access: Documents can be based on navigational access to the elements, access by queries or a combination of both if an OODBMS is used for the persistent data management.
- Activation and deactivation policy: Documents may implement different activation policies according to the needs of the application.
- Clustering: Elements of a document may be clustered within one or more segments or another policy may be used.

Documents can therefore vary greatly in their properties. To provide document classes for every reasonable combination of properties would lead to a large number of document classes and the

class library would be blown up with many similar classes. ETDB++ therefore provides a few general and some more specialized but often used document types. The general document classes implement general document concepts and they implement mechanisms which facilitate the derivation of other documents with different properties.

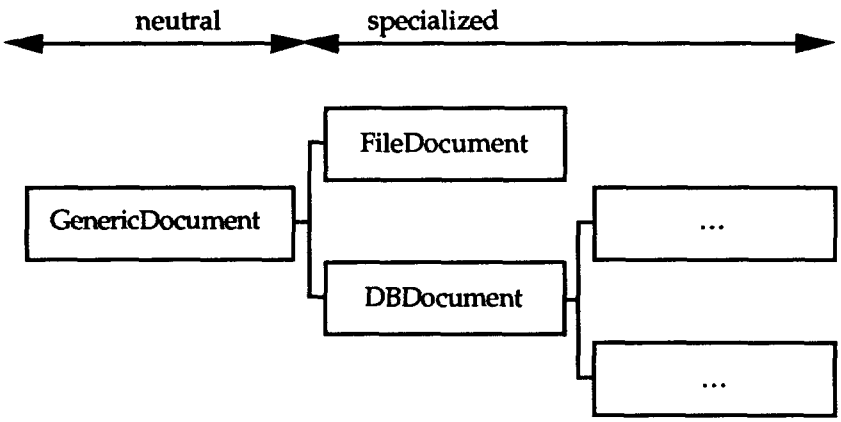

Fig. 6.4: Document types in ETDB++

The base class of all document classes is the class GenericDocument. It is an abstract class and defines the basic protocol of document classes to manage a group of objects and to control the windows, views, etc., in which the objects are displayed. No assumptions are made with respect to management of persistent data, concurrent activation of documents etc.

For the management of persistent data with files, the class FileDocument is provided. It implements the same functionality as the current document class in ET++, i.e., a document is a closed group of objects edited together. It allows the concurrent activation of multiple documents and the persistent data is stored in one file.

The class DBDocument implements the basic database document concept. It allows the concurrent activation of multiple documents and shared groups of objects. It assumes activation of its elements on demand and deactivation at end of transaction. It allows navigational access to its elements, access via queries or a combination of both. It assumes no particular clustering policy.

To facilitate the management of its components all active objects can be registered with their document. This information can be used to implement specialized deactivation policies in the case of shared objects.

For documents which rely on navigational access to their components another registration facility to permanently register its components is provided. Because the document knows which objects will be accessed it can acquire all the needed locks in one interaction with the OODBMS and it can ask to activate all or subgroups of its elements in order that the OODBMS can optimize network communication, buffer management and disk access.

6.1.2 Solutions to problems resulting from the architecture and the design of the OODBMS

Common problems

Problem of object faulting

If persistent objects must be connected to their nonpersistent environment to work properly one cannot rely completely on object faulting. Therefore it is suggested in ETDB++ that documents manage activation and deactivation and that all objects are registered at activation time with their document. The registration can then be used to trigger the proper installation. The basic database document therefore provides the necessary protocol needed to install objects properly during registration.

Variable length objects versus fix length objects

Many classes in ET++ rely on variable length objects. It is possible but not comfortable to integrate with a OODBMS only providing fixed length persistent objects but it is easier and more convenient to integrate with a OODBMS supporting variable length objects.

In the following sections, the paper discusses problems and solutions for OODBMS based on persistence by inheritance and persistence by allocation separately because the same problems usually have different solutions for the different systems or because the different systems have different problems.

Problems and solutions for OODBMS with persistence by inheritance

Persistent objects have not the same base class as transient objects

A good way to solve this problem is to provide a persistent object class which is substitutable with instances of the root class of the GACL (in ET++ Object) and which implements the basic functionality of the root class for persistent objects. Further persistent object classes are derived from this new persistent object root class. The persistent object classes derived from this classes are then also substitutable on the base of the root class of the GACL.

This can be achieved by implementing a forwarder class and a class which combines the forwarder class and the base class of the persistent object classes. The forwarder object is used whenever the persistent object should be substitutable with instances of Object and it simply forwards all message calls to the combination object. The combination object either calls back the method of the superclass of the forwarder object or deals itself with the message call (for a detailed description of the technique see [Schm90]). The combination class can be implemented with multiple inheritance or with composition.

Class libraries and OODBMS provide collection classes

In the best case one would have only one efficient set of universally applicable collection classes. One way to achieve this would be to simply not use one set. But because the features of the two sets are quite different one would at least loose a lot of possibilities or the solution simply wouldn't work, because, for example, the set of collection classes used couldn't cope with

transient and persistent objects. The goal for this project is therefore to find a solution which comes as close as possible to one set of collection class solution.

In the case of OODBMS based on persistence by inheritance it is possible to adapt the protocol of the OODBMS collection classes to the protocol of the collection classes of the GACL. This is achieved by implementing new persistent collection classes which use as their representation an OODBMS collection and whose protocol is as similar as possible to the already defined protocol of the GACL collection classes. This simplifies the use of collections, because programmers have to learn only one protocol for persistent and nonpersistent collection classes. This solution leads to substitutability with instances of Object but not with the collection classes of the GACL because the new persistent collection classes are derived from the new persistent object root class.

To achieve also substitutability with transient collections one has to apply the technique explained above for the problem of different root classes with the difference that the forwarder class is derived from the collection class for which a persistent counterpart is to be implemented. To prevent problems with the insertion of transient objects into such a persistent collection one has introduce runtime checks in the insertion methods.

The integration could be further improved by enhancing the transient collection classes so that they have the same query facilities as persistent collection classes. However, this would probably require a major effort.

Drawback of the integration is that persistent and nonpersistent versions exist.

Already defined classes should have persistent versions

For some classes of the GACL persistent counterparts should be provided. In the case of ET++ these are about 15 classes. Examples are text classes, event handler, and graphical object classes. The technique of forwarder and combination classes can also be applied here in the same way as with collection classes.

Problems and solutions for OODBMS with persistence by allocation

Class libraries and OODBMS provide collection classes

In the case of OODBMS based on persistence by allocation one can derive new classes from the base collection class of the GACL. These classes use the collection classes of the OODBMS as representation and act as protocol converters. This leads to a duplication of some classes because the GACL and the OODBMS both provide, for example, a class Set. The new collection classes also provide query facilities while the collections of the GACL provide only iteration. The collections are substitutable on the base of base class collection of the GACL. If duplicate collection classes, as for example Set, should be substitutable one would have to implement a new abstract class for each collection class and derive classes with different representations or parameterize the selection of the representation.

Eventually, if the collection classes of the OODBMS are efficient enough, one could simply reimplement the collection classes of the GACL with collections of the OODBMS as their representation.

Problem of cascading instantiation

There is no other way to cope with this problem than to modify all constructors in order that they can take an additional parameter to indicate where the object should be allocated. Moreover all cascading instantiations have to be modified to pass the additional parameter.

Which approach for persistence is preferable?

The integration is possible with both types of OODBMS. In systems based on persistence by inheritance one has to provide persistent counterparts to the collection classes and about 15 other already in the GACL defined classes. The integration is somewhat awkward in the case of the collection classes, probably because nonpersistent collections have no query facility and because of the duplication of the collection classes. An advantage is that the integration doesn't require any changes in the GACL as long as the collection classes are not changed for a query facility.

The integration with a OODBMS based on persistence by allocation is easier to accomplish because for already defined classes it is easily possible to allocate persistent instances. The integration of the collection classes could be smoother than for OODBMS based on persistence by inheritance if they are efficient enough to be used in place of the current collection classes of the GACL. A disadvantage of systems based on persistence on allocation is that one has to change the GACL in many places to cope with the problem of cascading instantiations.

Overall, the integration of the GACL and a OODBMS leads to a larger persistent object class library with more specific and meaningful persistent object classes.

6.2 Enhancement and extension of functionality of the OODBMS by exploiting GACL services

In ETDB++ the dialog facilities and the idle time of an application are used to enhance and improve the functionality of the OODBMS.

Use of the dialog facilities

The GACL supplies all necessary facilities to construct forms and in database application persistent objects often are displayed in forms. ETDB++ therefore provides a standard form for persistent objects. In the case of OODBMS with persistence by inheritance the standard dialog is part of the protocol of the root class of all persistent objects classes used in conjunction with the GACL. In OODBMS with persistence by allocation a class with such dialog properties is defined. Classes which should have this dialog facility are then derived from this class. In order to allow more flexibility the standard dialog facilities of a persistent object implement the methods needed to display the object and to cope with user events. The dialog layout is not hardcoded but determined by a policy object which can be exchanged at runtime.

Use of the idle time of the GACL

In many interactive application the computing resources are not consumed all the time in processing user interactions. The application is quite often idle while the user is thinking. Therefore one could take advantage of low activity time (think time) to do housekeeping tasks without interfering with the user activity. The generic application class therefore provides a service to manage tasks which are carried out during idle time, the so-called "idle tasks". A task may be installed in a task list. The application activates one task after the other. Tasks are not preempted. The task service assumes that tasks are cooperative, i.e., they stop execution after a short time. Tasks themselves are objects which have a protocol to start a task, to resume the task and to terminate the task.

One example for an idle task is incremental synchronization of the object cache with permanent store. During idle time changed objects may be written to permanent store. This reduces deactivation time and transaction commit time. Another task could be prefetching objects if it is known or highly expected that a user will access some objects in the near future.

The task system could be further improved if the operating system supports nonpreemptive threads. The complete GACL could then be based on these threads and the idle tasks would be only a special case of other threads of a application.

6.3 Using a priori knowledge to increase performance of the interaction with the dbms

The performance of object activation/deactivation is heavily dependent on the distribution of objects in permanent storage and on the order in which the objects are accessed. If queries are used to access objects a query optimizer hopefully tries to figure out an efficient order. If navigational access is used the optimizer is no help. With object faulting it may happen that for every fault another page or another segment must be accessed. However, if an OODBMS is asked to load a group of objects it could optimize the disk access, e.g., an elevator algorithm could be used, and it could optimize the buffer management. Even better performance results if the OODBMS is asked to load a whole segment.

There are also different access patterns. In the document editing style a bunch of objects is loaded, then these objects are manipulated for a longer time and then all the objects are written back to disk. Therefore, the time to load the objects and the time to store them on disk should be minimized. In the bank teller style, a few members of some big collections and some objects connected to them are loaded for a short time. It is therefore often not reasonable to load the whole segment, but the time to search the objects in the collections should be minimized. In the report access pattern often a high percentage of objects of one or more collection are accessed for a short time and some aggregations are computed.

Some of this knowledge can be used during physical database design to decide what attribute to index and which objects to cluster. The registration of the active elements and the permanent registration the elements of a document can be exploited in the activation/deactivation policy of a document to give the OODBMS hints for activation and deactivation of these elements.

However, some of this knowledge could also be used to build specialized documents. A programmer can simply use such a document if the problem in question requires the activation/deactivation pattern, the clustering policy, and other characteristics assumed for the implementation of the document.

An example of such a predefined document is the class SimpleDBDocument which implements an abstraction very similar to the FileDocument but for persistent data management with an OODBMS. It presumes that at most one document is concurrently active, that the document relies at first on navigational access and that the documents don't overlap. If the OODBMS supports clustering of objects all elements of such a document are clustered in one or more segments which don't contain objects of other documents. Activation, deactivation, and locking of such documents is very efficient, because whole segments can be locked, activated, and deactivated. In contrary to the FileDocument where multiple people could inadvertently change a file at the same time, the transaction mechanism of the OODBMS prevents the concurrent write access to such a document.

7 Conclusions and future work

This project shows that with the current state of technology only the integration of a OODBMS and a GACL supports applications with a modern graphical user interface. The integration of a OODBMS and a GACL faces several obstacles but viable solutions to overcome the hindrances are available. The integration also allows to improve the functionality of the OODBMS via use of services of the GACL and by taking advantage of a priori knowledge about interactions of the GACL with the OODBMS.

In future work one should consider a transaction concept more suited to engineering applications and the integration of versions. It would also be desirable to find an integration solution that works equally well with OODBMS based on persistence by inheritance or persistence by allocation. One approach would be to develop a generalized persistent object model which then would be mapped to the different database models. Eventually, high level tools which take advantage of the new GACL/OODBMS combinations should be built.

Acknowledgements

This work benefited very much from the helpful comments and benevolent critique of Judy Cushing, Belinda Flynn, Dave Maier and Jon Walpole.

References

[ACIU86] ACIUS, *4th Dimension*, ACIUS, 1986.

[Alta89] Altaïr, *The LOOKS User's Manual (Prototype Version 1.0)*, Altaïr, Le Chesnay, 1989.

[Appl85] Apple Computer, *Inside Macintosh Volume I*, Addison-Wesley, Reading, MA, 1985.

[Comp86] A. Computer Inc, *Open Dialogue*, Apollo Computer, Inc., Chelmsford, Mass., 1986.

[Deux90] O. Deux, "The Story of O2," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, Mar. 1990, pp. 91-108.

[Flyn90] Flynn, Belinda B., "*OODBMS Support for the Design and Management of Object Displays*", Unpublished manuscript, Oregon Graduate Institute of Science and Technology, Nov.1990.

[Gold83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, Reading, 1983.

[Gree85] M. Green, "Report on Dialogue-Specification Tools," In *User-Interface Management Systems*, G. E. Pfaff, ed. Springer-Verlag, New York, 1985, pp. 9-20.

[Huds88] S. E. Hudson and R. King, "Semantic Feedback in the Higgens UIMS," *IEEE Trans. on Software Engineering*, Vol. 14, No. 8, Aug. 1988, pp. 1188-1206.

[King89] R. King and M. Novak, "FaceKit: A Database Interface Design Toolkit," In *Proc. of the Fifteenth Int. Conf. on Very Large Data Bases (August 22-25, 1989, Amsterdam, The Netherlands)*, P. M. G. Apers, ed. Morgan Kaufmann Publishers Inc., Los Altos, 1989, pp. 115-123.

[Laen89] E. Laenens, F. Staes, and D. Vermer, "Browsing à la carte in Object-Oriented Databases," *The Computer Journal*, Vol. 32, No. 4, 1989, pp. 333-340.

[Lint89] M. A. Linton, J. M. Vlissides, and P. R. Calder, "Composing User Interfaces with Interviews," *IEEE Computer*, 1989, pp. 8-22.

[Maie87] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS," In *Research Directions in Object-Oriented Programming*, B. Shriver, ed. MIT Press, Cambridge, 1987, pp. 355-392.

[Myer89] B. A. Myers, "User-Interface Tools: Introduction and Survey," *IEEE Software*, Vol. 6, No. 3, Jan. 1989, pp. 15-23.

[Myer90] B. A. Myers et al., "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment," *Submitted for publication*, Jan. 90.

[Olso87] D. R. Olson, "ACM SIGGraph Workshop on Software Tools for User-Interface Management," *Computer Graphic*, April 1987, pp. 71-147.

[Onto88] Ontologic Inc., *Vbase+ Functional Specification*, Ontologic Inc., Billerica, 1988.

[Plat89] D. Plateau, R. Cazalens, and B. Poyet, *A customizable abstract I/O server for complex object edition*, Altaïr, Le Chesnay, 1989.

[Purd87] A. Purdy, B. Schuchardt, and D. Maier, "Integrating an Object Server with Other Worlds," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, 1987, pp. 27-47.

[Ross87] M. B. Rosson, S. Maass, and W. A. Kellogg, "Designing for Designers: An Analysis of Design Practices in the Real World," In *Proceedings SIGCHI+GI 1987*, ACM, New York, 1987, pp. 137-142.

[Schm86] K. J. Schmucker, *Object-Oriented Programming on the Macintosh*, Hayden Book Company, New Jersey, 1986.

[Schm90] D. L. Schmidt, *Persistente Objekte und objektorientierte Datenbanksysteme: Konzepte, Architektur, Implementierung und Anwendung*, Zürich, 1990.

[Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Readings, 1986.

[Sun 88] Sun Microsystems, *SunSimplify Overview*, Sun Microsystems, Mountain View, 1988.

[Webs89] B. F. Webster, *The NeXT Book*, Addison-Wesley Publishing Company, Inc, Reading, 1989.

[Wein89] A. Weinand, E. Gamma, and R. Marty, "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework," *Structured Programming*, Vol. 10, No. 2, 1989, pp. 1-25.

[Youn89] D. A. Young, *X Window Systems Programming and Applications with Xt*, Prentice Hall, Englewood-Cliffs, New Jersey, 1989.