Preliminary version. Final version in IEEE Software, 1998, March/April pp. 76-83.

97-03-22 21:07
oo-real.doc

Object-Oriented Systems in Real Life

Søren Lauesen

Copenhagen Business School, Howitzvej 60
DK-2000 Frederiksberg, Denmark
E-mail: sl.iio@cbs.dk

Abstract
Object-oriented development is expected to provide many benefits, but observations of industrial practice show that there are many problems to overcome. This paper discusses two issues: (1) The architectural issue of how to connect database, application, and screen objects. (2) To what extent expected benefits such as seamless transition from analysis to design, improved usability, etc. can be obtained. The architectural issue appears to have a significant influence on the expected benefits.

The paper is based on studies of about seven experienced, but very different, development teams, plus a lot of casual observations. I will present the three basic architectures that I found in practice and discuss their advantages and problems. Surprisingly, only one team had a solid architecture that solved most of the problems.

1. Background

The term "object-oriented" has been used as a catchword in recent years. We hear about OO user interfaces, OO design, OO languages, etc. Managers have heard about all the benefits of object-orientation: better user interfaces, reuse of code, cheaper development, and so on. In spite of the claimed advantages, little detail has been published about real-life object-oriented systems.

In the early eighties, I was leading an object-oriented project. Since then, I have been following the spread of OO and often wondered what benefit OO actually gave, particularly in business applications. Some developers of business applications have told me in private that their managers expected great benefits, but they could not themselves see the advantages. In contrast, developers of technical systems could tell about real benefits after some trials.

As a response to these comments, I published a critical article in the Danish version of ComputerWorld. It sparked a heavy debate and brought me into contact with many developers claiming they had successful OO projects. Typically they had more than five years of OO experience. Through many interviews and studies of about seven different development groups, I have tried to map the difficulties and identify the current best practice.

2. Object-Orientation

Object-oriented development starts with an object-oriented analysis where the problem domain is modeled as collections of objects. During design and implementation these objects are transformed into the objects that make up the actual computerized system [1, 2].

Formally, an object contains *data* and *operations* that operate on those data. Objects cooperate by sending messages to each other, i.e. call the operations in other objects. An important rule is that the data in an object is only accessible through the operations. No object can access another obejct's data directly.

In a truly object-oriented system, data and functions exist only in the form of objects. We can distinguish two kinds of *degenerate* objects: (1) An object without any data - this is the same as a traditional subroutine library. (2) An object with only trivial operations to retrieve data fields, update data fields, create and delete the object (CRUD operations) - this corresponds to a traditional data structure where the data is visible to the entire system. If a system is composed entirely of degenerate objects, we would not call it a truly object-oriented system.

As explained below, the business applications I observed turned out to consist mainly of degenerate objects.

Object-oriented systems are expected to provide many advantages compared to traditional systems [1, 3, 4]: Users should find the whole development process more understandable because the objects model the real world; there is a seamless transition from analysis to final implementation; the user interface should be better because the GUI objects on the interface correspond to the user's objects; the system is more easy to maintain because objects can be modified without involving the entire system; objects can be reused in other applications because they are encapsulated and provide a general "service".

My studies show that these expectations are far from being fulfilled. Business systems in particular cause troubles. There are several reasons for this:

One reason is that it *is* difficult to compose a real-life system of objects that can handle the screen, connect it to some form of a database, and also handle the semantics of the application. This is the architectural problem and current literature has very little to say about it.

Another reason is that the expected benefits are not realistic. At least they require something more than just "object orientation".

Below, we will first discuss the architectural problems and then the expected benefits.

3. The Implementation Layers

In order to compare the various systems I have seen, I will use the 3-layer model shown in Figure 1. The figure illustrates the model with a simple example, an order processing application. The figure does not show a specific implementation, but a joint collection of objects from all the architectures I met.

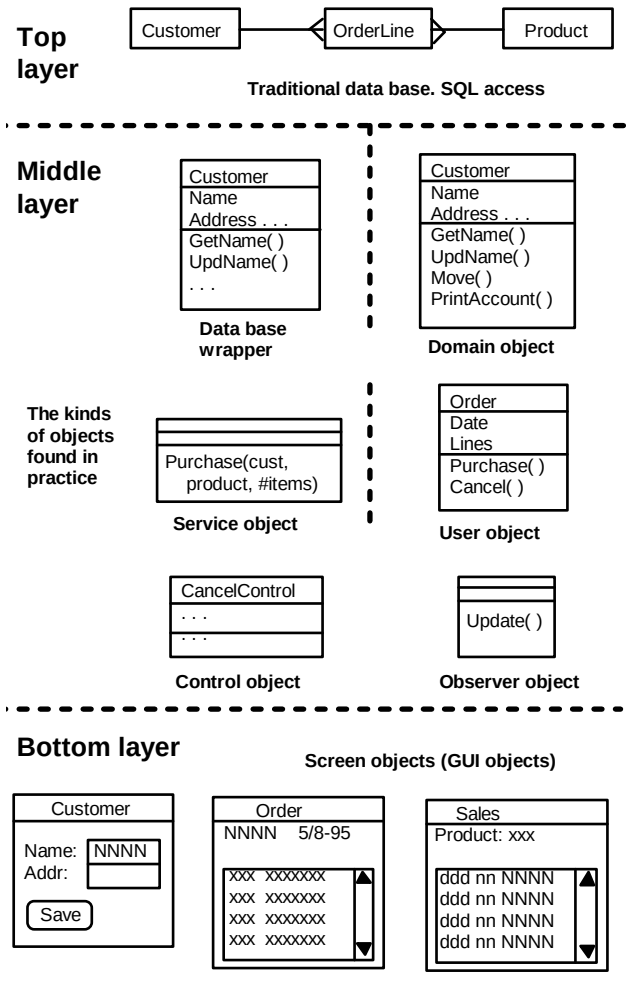

Figure 1. Three-layer model of an object-oriented system. The top and bottom layers are given by the platform. The middle layer illustrates the various kinds of objects found in practice.

The top layer is a *traditional database*, which typically is accessed through SQL-queries. In some technical systems (embedded software, etc.) this layer is absent. In all the business applications, this layer existed, and I have yet to see systems using an object-oriented database. The figure shows part of a database for order processing. It contains customer records, product records, and order lines that describe products purchased by customers. (Order records will exist too, but have been omitted here for simplicity.)

The bottom layer contains the *screen objects* that the user sees. These objects comprise GUI windows, fields, guiding texts, lists, scroll bars, menus, buttons, graphical curves, etc. The screen objects are arranged in a nested fashion (as aggregates). The window objects, for instance, contain fields and lists, etc. Since GUIs are object-oriented by nature, the screen objects actually exist in the implementation whenever a GUI interface is used. Several developers mentioned that the need for object-orientation in the user interface was the main reason for using object-oriented development.

The figure shows a customer window, an order entry window, and a sales window with a list of purchases for a given product. Note that the order lines are reflected in both the order entry window and the sales window. Similarly, customer names appear in all three types of window.

The middle layer forms the connection between the database and the screen windows. The figure shows the six kinds of middle-layer objects observed in practice. In real-life systems we find only some of them (typically between one and four kinds). The six kinds are:

Database wrappers that mirror the traditional database. A customer object, for instance, contains name, address, etc. for a specific customer. It has only simple operations that retrieve or update customer fields. There are also operations to create and delete customers.

Wrapper objects get their data from the traditional database and can write modified data back to the database. They are convenient as buffers for fast updating of the screen objects. Note that wrapper objects are degenerate objects since they do not contain high-level operations like Move or PrintAccount.

In systems without a traditional database, **Domain objects** contain the database. A domain object is usually more complex than a database record, and it can also contain high-level operations. Domain objects could correspond closely to the real-world objects determined during analysis [2, 5].

Service objects provide high-level operations, but do not contain data. They occur in many implementations but are degenerate objects, much like ordinary procedures or sub-routines. As an example, we could find an operation for registering a purchase. It would take three parameters: the customer, the product, and the number of items purchased. When called, this operation would create an order line object and link it to the customer and the product. It might also perform various checking to ensure consistency of the database. In Section 5.1, we will discuss alternative places to put the Purchase operation.

User objects mirror the screen windows. They contain the data shown in the window and have operations corresponding to the buttons and menu items available to the user. The Purchase operation would belong quite naturally to a user object representing the screen. User objects could be considered "logical windows", i.e. slightly abstract versions of the real window. For instance, they might hold a long list of data, whereas the real window only shows part of the list with an associated scroll bar. User objects correspond to what Rumbaugh calls "application objects" [2, 5].

When a field in the database is changed, **observer objects** distribute update messages to all screen objects showing that data. When receiving such an update message, the screen object may update itself to show the new database value. Gamma et al. call this the Observer Pattern [6, pp. 293 ff.].

In the example, three observer objects might distribute changes of the customer name to the three windows. Data values can also be reflected in more subtle ways on the screen, for instance as a total of many fields, graphical curves, graying of buttons in certain system states, etc. Observer objects can distribute update information to such screen objects as well.

Observer objects contain only a create, a delete, and an update operation. So they are also quite degenerate objects.

Control objects provide functionality closely associated with the screen objects. They could handle user actions when the user edits a field, pushes a button, etc. [7]. The actual GUI platform has a major influence on the choice of control objects. Some platforms do not require control objects at all, since the necessary functionality is specified as part of the screen objects (e.g. Visual Basic). Due to platform dependency, I will not consider control objects further, but assume that their functionality is part of the screen objects.

4. Actual Systems

How do actual, completed systems relate to the 3-layer model? I have found three basic architectures: Two for business applications and one for technical systems. Actual systems may deviate somewhat from the basic architectures.

For each architecture I will discuss standard issues it has to deal with, such as commit of transactions, consistency of the database, etc.

4.1 Simple Business Application Architecture

These systems have a traditional database. The screen objects are for instance written in Visual Basic. The middle layer consists of a buffer for each window, and the buffer is updated by means of queries to the database (Figure 2). According to the three-layer model, the buffers are degenerate user objects, i.e. without high-level functionality. Apart from the screen objects, these systems are not truly object-oriented.

Although the developers sometimes claimed that the development was object-oriented, this is not visible in the final implementation. During analysis and part of design, domain objects were considered, but their operations did not survive to implementation. The user functions found in the screen objects rarely correspond to such domain operations but reflect simple database operations.

On the other hand, this architecture is much simpler to implement than the other two. However, as we will see, it has difficulties handling two or more windows that share some data.

Commit: When will the data shown in a window be written back to the database? With the simple architecture, the answer is easy: When the window is closed or at a specific user request. The buffer is simply written back through one or more SQL-queries as a single transaction.

Consistency: The database must of course be kept consistent.

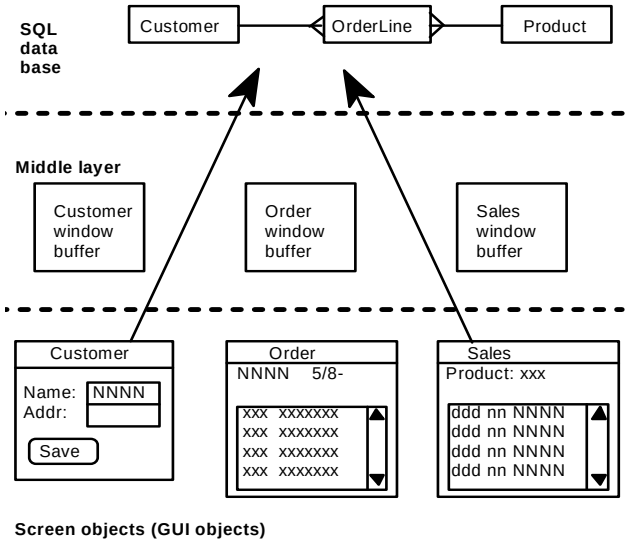

Figure 2. The simple business system architecture. The middle layer consists of degenerate user objects: a buffer for each window. The arrows show calls between objects. Notice the one-way calls.

With the simple architecture, some fields may be stored in more than one buffer. If the user edits such a field and closes the window, the database will be updated, but the old field value remains in a buffer for another window. When that window is closed, the old value may erroneously be written to the database. The typical solution is to restrict the dialogue so that only one window allows editing of the field, or so that only one window can be used at a time.

Data retrieval: How is data for a window retrieved? When the window is opened, the data can efficiently be retrieved into the buffer by means of one or a few SQL-queries.

Distribution: Efficient task support often requires that the same field is shown in more than one window. If the user edits such a field under the simple architecture, the changes are not reflected in the other windows. The usual solution is to provide a "refresh" button or menu point in each window, and leave it up to the user to ask for updating of the window if desired.

Call-back: In the other architectures, the call pattern between the objects can be complex with risks of endless recursion. But in the simple architecture, all initiative goes from the screen towards the database, so there is no call-back. This simplifies programming.

4.2 Complex Business Application Architecture

These systems use a traditional database combined with wrapper objects that hold data currently shown in some window (Figure 3). Observer objects link the wrapper objects to the windows that show their data. Note that a database record is handled by only one wrapper object, but may be shown in several screen objects at the same time.

The middle layer also holds service objects that perform high-level functions. This is convenient because high-level functions tend to refer to several database objects with a loose coupling to all of them (discussed further in Section 5.1). The screen objects retrieve data from and store data into the wrapper objects, and call service objects to perform high-level functions.

Since wrapper objects, observer objects, and service objects are degenerate, the solution is not truly object-oriented. Rather, it corresponds to a traditional system where data and functionality are separated. In practice, there are deviations from the pure architecture, and we may find some service objects with data or some observer objects with non-trivial operations. However, they are exceptions to the general rule.

As we will see, this architecture is quite complex to implement. It solves the problems that the simple architecture had difficulties with, but creates other problems.

Commit: It is no simple task to determine when to write data back to the database. Assume that a user edits a field. The result has to be stored in the corresponding wrapper object, since that is the only place data is stored in this architecture. But when will the value have to be written back to the database? Immediately, or when the window is closed, or when the last window using that database field is closed, or at a specific user request? Usually the right time is when the window is closed, but there may be exceptions. Writing back all the window fields has to be done as a single transaction, but that is quite foreign to the distributed logic of an OO system. This complicates the logic in screen objects, observer objects, and wrappers.

Consistency: Since each piece of data is stored only in the proper wrapper object, there is not the same consistency problem as with the simple architecture. Two different copies cannot accidentally be written back to the database.

Data retrieval: When a window is opened, each field will ask a wrapper class to retrieve the necessary data from the database. The wrapper class may determine that the field is available already in an existing wrapper object, or it may create a new wrapper object and retrieve data from the database through an SQL-query. This is clean and easy. However, when a window with for instance a long list of records is to be opened, an SQL-query is used for each record - or maybe for each field in the record. This can give serious performance problems, in particular if the database is remote. (In one case I studied, the database was on the other side of

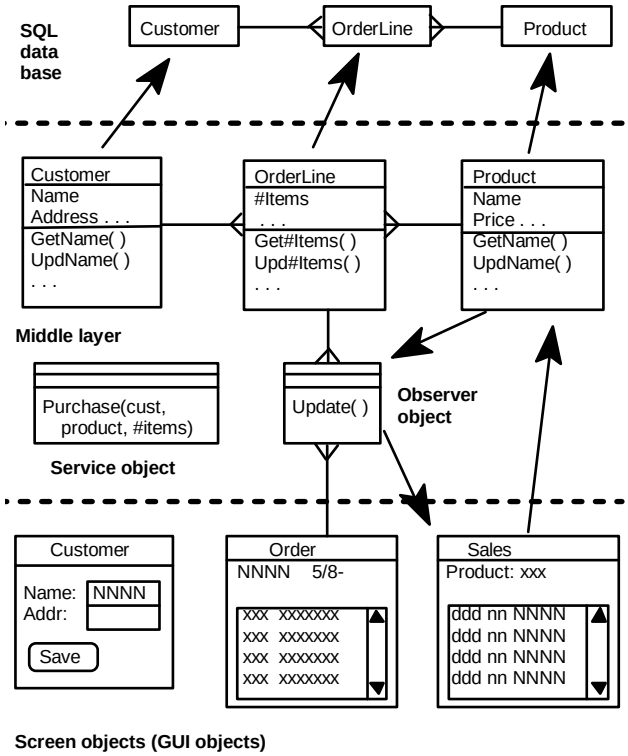

Figure 3. The complex business system architecture. The middle layer consists of degenerate objects. The database envelopes serve as buffers for screen updating. The arrows show calls between objects. Notice the two-way calls.

the globe, so the problem was serious.)

The proper solution would be to retrieve all the necessary data with one SQL-query, but that is foreign to the distributed logic of an OO system.

Distribution problem: In contrast to the simple architecture, the complex architecture offers a solution to the distribution problem: the observer objects. If the dialogue is just a bit complex, the same record might be reflected in several screen objects, and a single screen object might reflect data from several records, for instance in case of a field that shows the sum of a list of records. This calls for a many-to-many relationship between screen objects and wrapper objects. One purpose of the observer objects is to implement this relationship.

Special logic is needed for updating fields with aggregate data such as totals or graphical curves. Time-consuming updates are usually only made at user request.

An observer object should be able to connect any wrapper class with any screen object class. In C++ this gives severe binding problems, multiple inheritance, and leads to a huge class hierarchy in real systems. The problems and a complex solution have been described by Jaaksi [7].

In practice, developers make more or less ad hoc solutions to all these problems.

Interestingly, the only complete and general architecture I saw in practice, was based on SmallTalk, which does not have the binding problems of C++. The solution is developed and marketed by a small software house, ObjectDesign [8]. They have developed several large business applications by means of their architecture. SmallTalk has a built-in distribution/observer mechanism, MVC, which is inefficient for larger systems, so ObjectDesign built their own observer system. This architecture also had efficient solutions of most of the other problems.

Call back: Another problem is that the initiative goes both ways: The screen objects call the middle layer for retrieving and updating data, and the wrapper objects call the screen objects - through the observer objects - for distributing changes. This requires careful handling of concurrence to avoid deadlocks and endless recursion. Several developers had experienced problems in this area, and one group had given up solving the distribution problem for that reason.

4.3 Technical System Architecture

This architecture does not use a traditional database. Data are stored in domain objects that reflect the physical system controlled by the computer. Usually the domain objects are not simple record structures, but highly structured aggregates of data or sub-objects. Service objects may be added for convenience. Observer objects are often used to distribute data changes to the screen objects. Since the data is only found in the domain objects and not in a traditional database too, the domain objects are simpler to implement than wrapper objects.

The architecture is truly object-oriented. This is probably the reason why object-orientation has been quite successful in technical systems like complex equipment, process control, etc. To some extent analysis objects can be traced to implementation of the middle layer, although some developers said that there was little traceability.

As we shall see, the implementation of the architecture is quite complex if the distribution problem is to be solved.

Commit: When will the data shown in a window be written back to the domain objects? Usually as soon as the data has been edited. But even in technical systems there may be cases where a group of fields have to be committed at the same time. Typically, that is solved through a closed dialogue where no other updates of the domain objects take place meanwhile.

Consistency: Since the data is stored in only one place - the domain objects - there is no risk of two different *screen copies* being stored back to the domain objects. However, technical systems are often a kind of process control system, and that gives rise to another kind of consistency problem: Concurrent threads may try to update the same data at the same time, for instance trying to update the same counter.

This is solved in traditional ways, for instance by letting the domain object do the entire update through an asynchronous operation. The domain object is then a separate thread that can handle only one message at a time. Alternatively, locks, semaphores, etc. can be used, so that only one thread at a time is allowed to retrieve and update domain data..

Data retrieval: Data to be shown on the screen is simply retrieved from the domain objects. The performance penalty for retrieving the data in several small chunks rather than one big chunk is small compared to the time it takes to update screen displays.

Distribution: Observer objects are used to connect the screen objects to the domain objects, so that a change in a domain object can be distributed to the screen objects showing the data.

The solution is similar to the complex business applications and requires the same careful implementation. However, the technical architecture has to deal with an additional problem: In the complex business architecture, data in the wrappers can be addressed in a uniform way, because the wrappers are an extract of a relational database. What a screen object has to do, is to specify a wrapper class and a key for the record in that class. This makes all observer objects quite similar. In

contrast, data maintained by domain objects cannot be addressed in a uniform way, since the data inside a domain object can be structured in many ways with arrays and records in many levels. This means that there are even more observer object classes in the technical systems.

For the same reason, it can often be difficult to extend the system so that it shows data in ways that cross-reference data from different domain classes. It may be necessary to extract data from inside the complex domain objects, and new operations in the domain objects may be needed for that.

Call-back: The technical architecture has the same problems with two-way initiative as the complex business architecture.

4.4 Summary of Architectures, OO-Databases?

If we compare the three basic architectures, we see that the simple business architecture is not object oriented at all. Apart from that, its only weakness is that it has troubles handling data shared between two windows.

The complex business architecture consists primarily of degenerate objects. It is very difficult to implement, and only one team had implemented it fully. There are also performance problems when retrieving large quantities of data from a remote database.

The technical system architecture is truly object oriented, but it too is quite difficult to implement.

Most applications do not completely follow a single architecture. One business application, a financial information system with a world-wide database, followed the complex business architecture, but had also user objects with their own data and computationally complex functionality for analyzing trends and stock values. One would expect that this gave problems with commit, consistency, and distribution. However, the system's main purpose was not to update the database, only to retrieve data. For this reason, the commit, consistency, and distribution problems did not exist, and no observer objects were needed. The few updates that could be made were handled with ad-hoc logic.

It is reasonable to ask whether an object-oriented database would have helped. I did not see any examples in the cases I investigated, but an OO database would probably contain an implementation of the wrapper objects. A few groups had tried out an OO database, but had found performance problems with data retrieval. It is not obvious that OO databases would solve the problems with consistency, distribution, and call-back. Dana Moore [9] has a thorough discussion of the relation between SQL and OO databases.

5. Development And Usability Aspects

In this section I will discuss to what extent the object-oriented approach eased development and improved usability of the final system.

5.1 Object Orientation and Seamless Transition

In the technical systems there was often a clear trace from analysis to final implementation, but in the business applications this was rarely the case. Particularly the operations did not transfer in a seamless manner. While the technical systems had domain objects with non-trivial functionality, the final business applications consisted primarily of degenerate objects. I believe there are three causes for this:

(1) The technical systems model a physical world consisting of equipment with various parts. This can be reflected as aggregations of objects, where each object belongs to only one aggregate. Further, the interface to a physical part can map quite well into operations in the corresponding object.

In contrast, business applications handle data that do not aggregate naturally, but relate to each other in many ways. To the user, an order form is a physical entity and we might try to model it as an aggregation of an order heading and a list of order lines. But the order lines appear also in other "aggregates" like sales reports. This means that we have to model the basic entities as separate objects and link them to several other objects.

(2) Since we have to model the business data in this way, we have no natural place to put a high-level operation like Purchase. It could belong to the customer object, the product object, or the order heading object. As a result, it seems more convenient to avoid an arbitrary decision and instead put the operation in a service object, i.e. a degenerate object without data of its own.

Another argument for separating high-level operations from the objects is given by Maring [10], describing GTE's many years of experience with large, object-oriented systems. He explains that they did not succeed until they put control flow and business processes (i.e. high-level operations) outside class behavior. If built into the classes involved, it was impossible to get an overview of the control flow: it was like "reading a road map through a soda straw".

(3) Traditional developers of business applications report that information models (data models) transfer very well from analysis into the final system. In contrast, the functional aspects can be described in the analysis (for instance as data flow diagrams), but do not transfer well into the final system. If this observation has a counterpart in object-oriented development, it may be the reason why operations in particular don't transfer well to implementation. As a simple example, consider the operation

```
Customer.Move(newAddress)
```

This operation may seem natural during analysis, but at run time there is no such menu point as "Move". Instead, the user handles the operation by simply editing the address field of

the customer. In general, the user carries out most tasks through editing or other low-level database operations.

Does this mean that truly object-oriented business applications cannot be made? I found it striking that those developers with the most experience and many successful systems behind them, accepted the largely degenerate models as the best solutions for business applications.

5.2 User Involvement During Development

To what extent could users understand the object-oriented approach and contribute during development?

According to some approaches (e.g. Rumbaugh et al. [1]), the analysis and design start with object diagrams that use a notation rather similar to E/R-modeling (like Figure 3). All designers using this approach reported that they had tried to discuss the diagrams with the users, but the diagrams did not make sense to users at all.

In other approaches (e.g. Wirfs-Brock et al. [11]), the primary diagram is an object interaction diagram which shows the messages passed between objects. These diagrams have more of the flavor of traditional data flow diagrams. One group reported that this gave some basis for discussion with users, but users seemed to think of the boxes as functions rather than as sets of objects.

In general, what OO analysts call objects may not be natural objects to users. In our example, customers may be natural objects since they can perform something, but a user is confused if we talk about an order as something that can perform operations and has responsibilities. Rosson and Alpert discuss this in [12].

Some groups had experienced that initial user involvement was successful if based on traditional requirements specifications, including scenarios and task descriptions in plain text and on a rather high level. Users could readily comment on such descriptions and add further tasks to be supported.

Later user involvement was successful if based on prototypes, either as computerized prototypes or as simple paper mockups.

In conclusion, object-orientation itself did not ease user involvement, but traditional techniques - like scenarios and prototypes - did.

5.3 User Interface

Until now we have primarily looked at analysis and design of the internal system functionality. But the user sees only the screen objects. How were they developed? When discussing this issue, I will only look at the business applications, because I can see a general pattern there.

In most cases, the developers had no clear explanation of how the screen windows were chosen. The attitude was that the system had to show the contents of the database somehow.

In the actual systems, most screen windows were *database oriented*, i.e. they corresponded to simple database records or to simple records with an associated list of related records. The customer window of Figure 3 is an example of a simple record shown in a window, while the order window and sales report window are examples of a record with an associated list. These kinds of windows can easily be generated by CASE tools or database packages - based on the database structure. Baskerville [14] and Balzert [15] show a systematic way to do that.

Although this user interface is close to the data model found during analysis, it causes several problems to the user:

Few windows for frequent tasks: A consequence of database-oriented windows is that the user often needs to access many screens to carry out a frequent task. As an example, I saw a system for registering blood pressure and temperature of hospital patients. Since these measurements were two different object types, there was one screen for entering blood pressure and another one for entering temperature, although the two measurements usually were registered at the same time.

One way to remedy such problems is to design more complex windows where the necessary data for a task is collected. This requires screen designs that explicitly consider task requirements. Lauesen et al. [16] show a systematic way to do that. Since Jacobson's introduction of *use cases* (similar to

Time registration Init: MBH

Activity	Date	Hours
102 Lunch	290595	0.5
715 Design DXP	290595	4.0
812 Cust. meeting	290595	3.0
102 Lunch	300595	0.5
715 Design DXP	300595	3.0
808 Review SPA	300595	4.5
102 Lunch	310595	0.5
715 Design DXP	310595	5.0
808 Review SPA	310595	2.5

Time registration Init: MBH Week: 22 Year: 95

Activity	Mo	Tu	We	Th	Fr	Sa	Su	Tot
102 Lunch	0.5	0.5	0.5		0.5			2.0
715 Design DXP	4.0	3.0	5.0		3.0	4.5		19.5
808 Review SPA		4.5	2.5					7.0
812 Cust. meeting	3.0				3.5			6.5
901 Course				7.5				7.5
Total	7.5	8.0	8.0	7.5	7.0	4.5		42.5

Figure 4. An example of data shown in traditional list form, and the same data shown as a matrix that enables easy perception of patterns.

"tasks"), OO methods should have a potential for matching tasks and screen design [13]. Two groups systematically made complex windows for efficient task support, but they did it as prototype experiments in close cooperation with users, not based on an OO method.

Understanding the database: In order to perform unusual tasks, the user has to navigate through the database-oriented windows. It requires a good understanding of the database. But as we saw with user involvement, it is not easy for users to understand the various types of records (objects) and their relation. So in this respect, the systems were no better than traditional mainframe-based business applications.

Understanding the database operations: In spite of style guides, etc., users have surprising difficulties finding out how to use the database functions. For instance, when is the database updated? When you have edited the field, or when you close the window, or when you use the Update function? This problem is closely related to the architectural commit problem.

A special concern is the search mechanism necessary to retrieve data from large databases. Typically, the user enters some search criteria and sees a list of matching records in a window. But if the search criteria do not stay on the screen close to the list, the user easily loses track of what is in the list. For instance, a bit later the user might believe that something is not in the database because it does not appear in the list. The problem is aggravated in certain systems that use the same screen fields for search criteria and results. The user easily loses track of the current mode: Do the fields show criteria or retrieved data?

Editing a list: Quite often, fields in a list cannot be edited, and editing must be done in single-record windows. This restriction is related to the consistency problem discussed earlier, and also to limitations of standard classes to show and handle lists.

Easy perception of structure: The database-oriented screen design can show a lot of data in a condensed form as lists and fields, but it is difficult to perceive patterns and structure in the data. Nygren et al. discuss this in [17]. The object-oriented approach itself does not encourage such efforts. Only two of the business groups I studied paid attention to this issue and could produce advanced pictures.

Figure 4 shows a simple example of the list approach compared to an approach where data form patterns. The example concerns registration of time spent on various development activities and overhead activities. Each entry in the list shows the activity worked upon, the date worked, and the number of hours worked that date. In the matrix presentation of the same data, it is easy to see patterns. We can visually check that lunch is registered every day, that certain activities take a full day, or that they are worked upon every day. We can also easily check that a full day is registered, etc.

5.4 Modifiability and Reusability

It was difficult to get systematic information about maintenance and reuse. Most developers agreed, however, that the object-oriented design was much more complete than the traditional designs they had made. Some also believed that the program was easier to modify.

There was little experience with reuse. One group explicitly said that they had realized they could not reuse classes in other projects in the company. However, they believed that object orientation had allowed them to agree on concepts and terminology across many teams. Other groups were able to reuse the architecture they had developed and common parts of the wrapper objects. The ObjectDesign group could reuse all the classes implementing their architecture, and they market that package as a product.

6. Conclusion

Inspections of several industrial, object-oriented projects and talks with developers show that many of the expected OO-benefits are not obtained in current practice. Furthermore, there are serious problems finding a solid architecture and implementing it.

Why, then, do companies invest in OO? Usually, the reason is one or more of these:

- They hope that the expectations come true.
- They need GUI interfaces and client-server technology, and OO seems the only way to get it.
- They have never tried data modeling and become happy with the data modeling aspect of their chosen OO method. (Companies with data modeling experience consider OO more of a gradual evolution.)

What can be done about it? Much work has to be done in designing and implementing one or more solid architectures - or finding an existing good solution. It should also be realized that the ideal presented in the textbooks is not realistic and sometimes even harmful. Finally, it must be realized that OO alone does not provide all the expected benefits; other techniques and methods have to be used in addition.

References

1. Rumbaugh, J. et. al.: Object-oriented modeling and design. Prentice-Hall, 1991.

2. Rumbaugh, J.: Objects in the twilight zone. Journal of Object-Oriented Programming, June 1993, pp. 18, 20, 22, 24.

3. How to overcome the object technology learning curve. I/S Analyzer, Vol. 33, no. 8, August, 1994.

4. Henry, S. & Humphrey, M.: Object-oriented vs. procedural programming languages: Effectiveness in program maintenance. Journal of Object-Oriented Programming, June 1993, pp. 41-49.

5. Rumbaugh, J.: Modeling models and viewing views. Journal of Object-Oriented Programming, May 1994, pp. 14-20, 29.

6. Gamma, E. et. al.: Design patterns. Addison-Wesley, 1995.

7. Jaaksi, Ari: Implementing interactive applications in C++. Software-Practice and Experience. Vol.25 (3), March 1995, pp. 271-289.

8. ObjectDesign. Software house. Contact: Anders Bonde, obdeandb@inet.UNI-C.dk

9. Moore, D.: An Ode to persistence. Journal of Object-Oriented Programming, Nov/Dec 1996, pp. 29-34.

10. Maring, B.: Object-oriented development of large applications. IEEE Software, May 1996, pp. 33-40.

11. Wirfs-Brock, R., Wilkerson, B., and Wiener, L.: Designing object-oriented software. Prentice Hall, 1990.

12. Rosson, M.B. & Alpert, S.R.: The cognitive consequences of object-oriented design. Human-Computer Interaction, 1990, Vol 5, pp. 345-379.

13. Jacobson, I.: Object-oriented software engineering - a use case driven approach. Addison Wesley, 1994

14. Baskerville, R.: Semantic database prototypes. Journal of Information Systems (1993) 3, pp. 119-144.

15. Balzert, H.: From OOA to GUIs: The JANUS system. Journal of Object-Oriented Programming, Feb 1996, pp. 43-47.

16. Lauesen, S., Harning, M.B. & Grønning, C.: Screen design for task efficiency and system understanding. In S. Howard and Y.K. Leung (eds.): OZCHI 94 Proceedings, pp. 271-276.

17. Nygren, E., Lind M., Johnson, M. & Sandblad B.: The art of the obvious. CHI'92, pp. 235-239. ACM 0-89791-513-5/92/0005-0235.