CAP

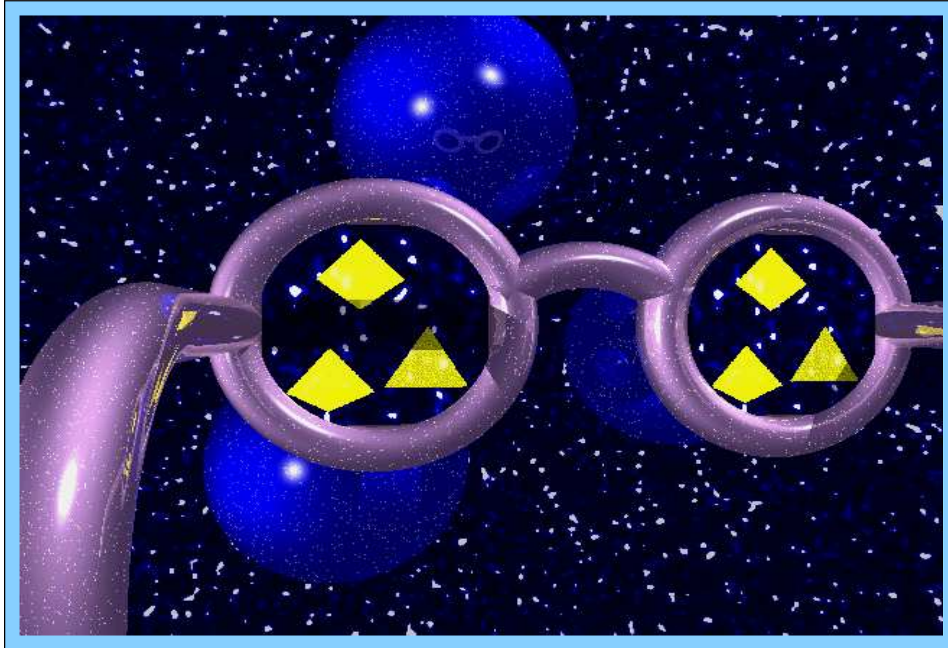
The CyberSpace Architecture Project

©1994,95 by Andreas Leue

Object Oriented Views

I Wear My Sunglasses at Night

This paper describes Object Oriented Views (OOV) and how to use them to build an integrated object oriented system which supports the views of different users. The concept of orientation not only to an object, but also to a subject is introduced. Finally the relation to the Cyberspace Object Architecture (COA) and other implementation issues are described.



An Object Oriented View (OOV) of some objects (blue balls).
From the subject's personal point of view they look like yellow pyramids.

1 Introduction

Stated in one word, OOVs are small, surveyable class and object models. The purpose of the concept of OOVs is to emphasize the role of such small standalone models.

OOVs are the interface between closed subsystems and the rest of the system, probably the whole outer world. Modeling these OOV interfaces gives a precise description of the coupling of the subsystems to their environment, which is a valueable base for porting, adapting or maintaining them.

Furthermore, OOVs provide a description of a system related to a users point of view, thus incorporating *subject orientation.* This gives the freedom to ignore uninteresting aspects and to use subjective ones.

An object oriented system with OOVs is not described by one hierarchically ordered class model. Instead, there can be one or more "central" class models, and several OOVs. The coupling between them is described by various relations, such as classical *inheritance*[1] relations, *dynamic inheritance* relations and, most flexible but cumbersome, *encapsulation* relations.

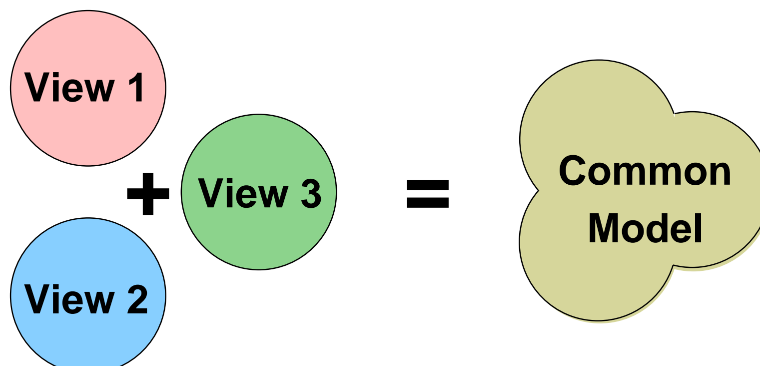
The emphasis of the subjective role of class models and arbitrarily coupled subsystems, is in contrast to hierarchical and somewhat monolithic class models more suitable to describe complex and open systems.

[1]The term *inheritance* is used intentional in this text, since they are only one kind of implementation of *generalisation* relations, namely static ones.

2 Origins of OOV

The next two sections give some information illustrating the background of OOVs.

2.1 Conflicts in Object Oriented Design

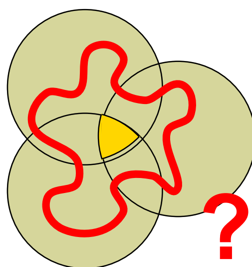


Integrating different views into one model.

The first section refers to object oriented modeling. To design a good model one has to analyze first the various needs of the users of the system. There may be several views depending on the point of view. These views have to be integrated into one model.

The figure above shows that in a somewhat schematic fashion. The term "view" is used here for requirements, classification, strategies etc. a user applies to a system. "Model" denotes an object oriented model used to describe the system, and a "user" is someone who has to deal with this model, as a real end user or as a programmer.

It is the art of designing to perform this integration satisfying, and it is possible to create good systems that way. But there is nevertheless a more general problem associated with. Consider the following figure.



How to define the boundary of the common model?

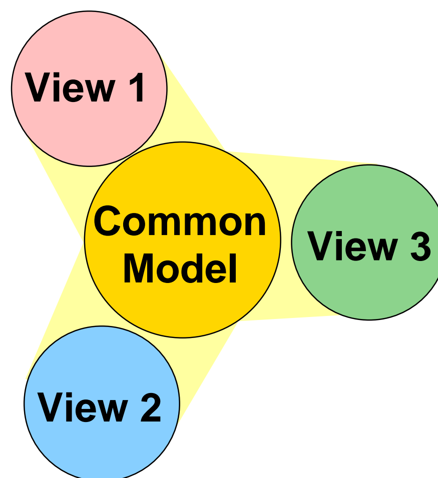
Given the three views, there are two natural limits between which the resulting model will be placed sensibly.

The first is the union of all requirements: a model incorporating all of them is certainly complete enough, but each user of the model has likely to deal with some requirements of other users, which are of no interest to him.

The second limit is the intersection of the requirements: in this case, no user will be confronted with any other users requirements, but the model is most likely incomplete.

It is diffcult to define the boundary of the model between these two limits. This is because completeness and ease of surveyance are contradictory requirements.

Taking other requirements such as maintainability into account leads to the object oriented design approach: to focus on the objects, their identity, states and behaviour. The result resembles the second approach above, but is more "objective". Since the focus is on the object properties, the *subjective* requirements are ignored. This loads a burdon on the user, who probably is not primarily interested in objectivity.



Design an object oriented model, but add subjective views.

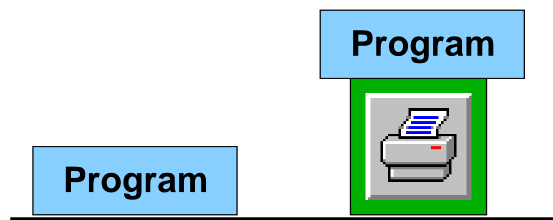
The figure shows a solution: perform not only a good object oriented design of the common objects, but apply the same method to model the views of the users and link the results together.

Each of the models, the common one and the views, are easy to survey, and togther they form a complete description of the whole system.

Note that there is another level of abstraction introduced: it is given by the last figure and shows how the various models and views are related to each other.

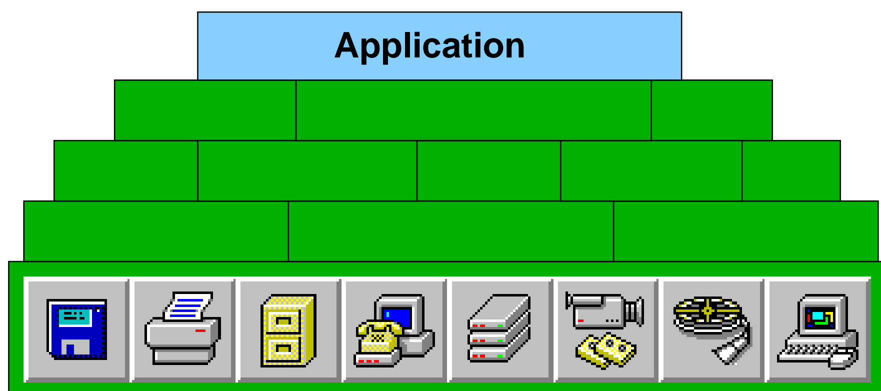
2.2 The Notion of Applications

The second section describes the evolution of the concept of an *application*.



In the beginning, there were only programs.

In the good old days, there were no applications. There were only *programs*, and if you had real luck, you could find a *device driver* or some small *system service*.



On the solid ground of hardware, layers of system services founded a base for applications.

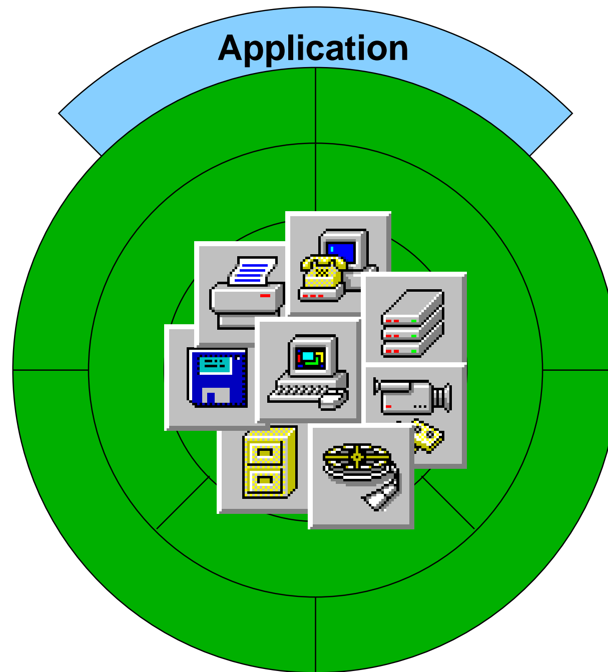
Next, the decade of the solid founded systems arised. There were *operating systems*, based on the solid ground of the hardware. They were arranged in *layers*, the higher ones providing more and more sophisticated services. At the end, it looked like there will be nothing left to do behind the highest levels, where now the *applications* resided.

There were some problems, of course, if the layers changed or grewed. Nevertheless, they did change and grew, and the solid ground turned out not to be solid at all. Even worse, the need for moving an application to another system - bad idea - arised.

But a solution was easy to find: put yet another layer on the top.

This last layer had to have the special property not to change. This lead to the examination of the common and invariant properties of layers serving similar purposes. A good example are *Graphical User Interfaces (GUI)*.

An improvement of the model of layers on the solid ground was the shell model, which is shown in the following figure:



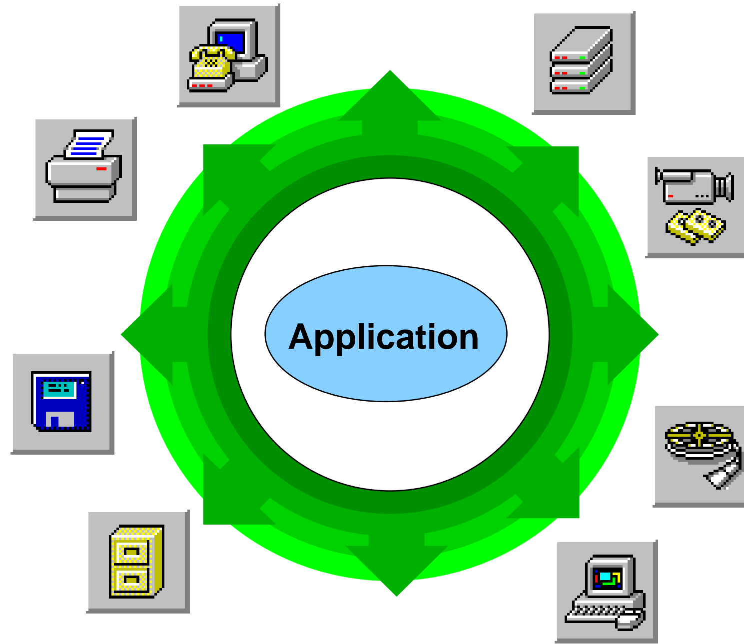
The solid hardware ground shrinks to a system kernel.

This shell model is an improvement since the notion of a kernel is a more realistic picture than a solid hardware ground. But still, the system services are the central entity and the area of the applications is undefined and somewhat only an *add-on*. And of course, the systems change and grew and are far not something static like the term "kernel" suggests.

So, after the decline of the solid ground, the systems learned at least to move. Today, in the context of complex, open and world wide connected systems, the notion of changing systems further evolves to an undefined and unforeseeable environment. The systems are no longer static points to refer to.

This is no loss, since it forces one to concentrate on the tasks to perform and to ask again on what base to place the applications.

Consequently, let's reverse the last picture and put the application in the center:



An application in the open world.

The application is now the central instance, embedded in some layers screening it from the world's specific elements, a kind of *ex–capsulation*.

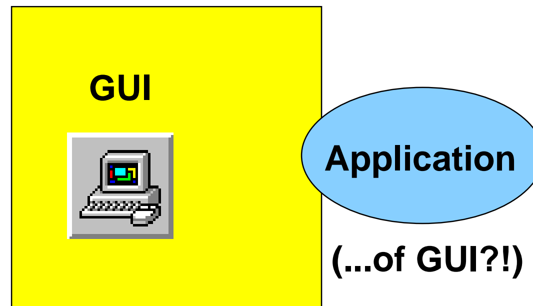
This reversion is not an academic one, but implies several consequences and advantages:

- The concentration on the application frees from the need to define general layers for all possible applications. An application can be embedded in a very subjective environment.
- The focus on the application implies to be best prepared for porting purposes, since ideally only the applications needs are described, and no implementation aspects.
- Starting from the applications needs, one can build more surveyable systems better suited for a given task.

This picture reflects an old approach of software engineering: to concentrate on the tasks to perform and the needs of "the application".

Applications and Objects

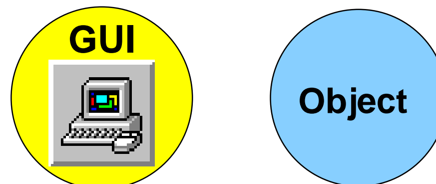
Another change can be observed: the replacement of "applications" with "objects".



A GUI system and an application of it.

Reasoning over the term "application", one can ask who or what exactly is applied to whom or what? The term is in fact perfidious, it states implicitly that a software application is only an application (verbal) of some system, which is the central and interesting part, as the figure above illustrates.

To describe evolving and big systems with many interacting and specialised components other terms seem better suited: cooperating components, service provider and clients, communication. And, most important: objects. The term "application" is no longer appropriate in such systems. The following figure shows this change.



An object using GUI services provided by another object.

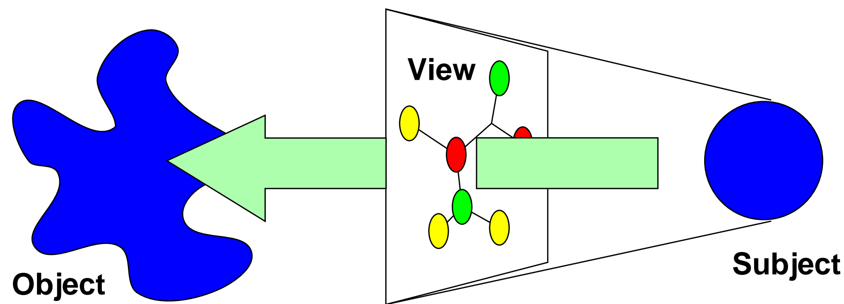
This also emphasizes the new role of "applications": systems designed to solve a specific task, which are no longer considered as simple supplements to the real systems, but as the central and finally solely important part. Consequently, it emphasizes the need of appropriate frames for such systems.

The Final Picture

Taking also into account that applications themselves become more structured, flexible, customizable and share more code with each other, the following picture may give a good impression of today's open, complex and worldwide connected systems:

Objects floating in today's environments.

3 Object Oriented Views



An object oriented view of an object from a subject's point of view.

OOVs are small surveyable class and object models. They represent a well suited interface allowing access to some system with regard to specific needs of a user. They are

Object Oriented in that they focus on the objects of the system, their identity, states and behaviour and the modeling technique used is an object oriented one; they are also

Subject Oriented since they respect an explicit user – the subject – of the model and reflect it's personal point of view of the objects.

OOVs do represent objects from the point of the viewer. [2] They encapsulate uninteresting aspects of these objects and allow to view and manipulate them appropriate to the needs of the users.

The link between OOVs and the rest of the system may be of any kind. They do not need to be placed in a hierarchical class model, but they can. There are three main categories of relations between OOVs and the environment:

Inheritance Relations connect the OOVs classes as bases to the model, using mutliple inheritance as a mean to create different interfaces to the same object.

Dynamic Inheritance Relations serve for the same purpose from a logical point of view, but do not require the base class to exist statically.

Encapsulation Relations are used if there is no clearly defined one-to-one relation or if the relation is not of a generalisation kind. They are most flexible but cause the most work, too.

For the first kind of relations there is support given by OOPLs, but for the second and third one there is none. To provide some support for these relations is a goal of OOVs (see **??**).

[2]From a system or implementational point of view they probably contain only interfaces to the "real" objects

3.1 Subject Orientation

OOVs represent a direct consequence of applying the principle of *Subject Orientation*.

Each subject (user) has a specific *point of view*, from which results a subjective view of the system. This view is the link between the subject and the system. It allows the perception and manipulation of the system in a manner which is well suited for the subjects needs.

Such a subjective view gives the freedom to ignore uninteresting aspects of the system and to add subjective ones.

3.2 Aspects

An *aspect* characterizes the point of view one can take on. Each OOV is created and designed under a given aspect, which results from the perspective of the viewer.

Since it may be useful to request services from objects at runtime with respect to the requester's point of view (e.g. creating a presentation of the object), this point of view has to be captured somehow and passed as a parameter of the request. This is the purpose of `Aspects`.

`Aspects` are not identical with subjects, and there is not necessarily a 1-to-1 relation, since `Aspects` may be more abstract. An `Aspect` may describe a group of points of view.

3.3 OOVs and Frameworks

This section explains the relation between OOVs and *frameworks*, since there are some similarities, but also differences.

Both OOVs and frameworks have in common that they capture a *whole set* of objects and classes and not only single entities. This is a useful and necessary extension of simpler class libraries since it allows to capture important aspects of overall system design in addition to the provision of building blocks.

But while frameworks provide also a configurable instantiation of using the system - an application of the system -, OOVs do only provide integrated objects and classes (primarily) without such an application skeleton.

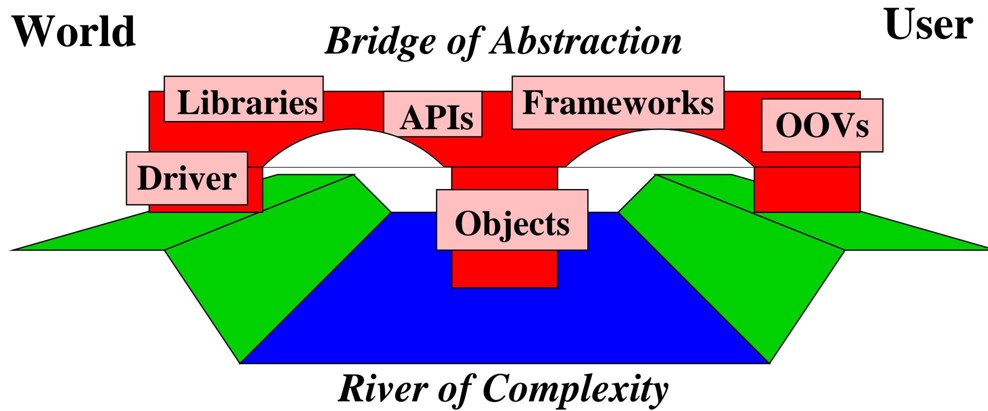
It may be a clear design approach of frameworks to seperate these two aspects: to build an explicit platform (like an OOV) and to place an customizable application process class onto it.

Seperating these aspects a user has more freedom in applying a framework. This is useful, since the process skeleton of a framework may in some cases be insufficient. Integrating the process with the environment and giving the user no chance to change the frameworks kernel implies the statement, that this framework is in 100 percent of applications correct and usable. Such a high percentage is most unlikely to be observed in the real world.

But there are further differences between OOVs and frameworks.

3.3.1 The Bridge of Abstraction

Frameworks are a continuation of class libraries in that they serve the purpose of providing (more or less) common functionalitites to a group of appliers. In contrast to this, OOVs focus on a concrete user and are applied in designing a concrete system. They serve as a building block on the other side of the *Bridge of Abstraction*, which leads over the *River of Complexity* as shown in the following figure.



The Bridge of Abstraction leads over the River of Complexity.
This simplified figure shows only the role of software.

Coming from the left side – a concrete part of the world – the bricks are abstracting from the details and become more and more general. Walking on to the right, there is a turn in this behaviour: the bricks become again concrete, but now with regard to the user.

It is important to notice this fact, since it implies the refusal of the idea of "general problem solvers", which can be used to solve every problem a user may have. "General" solutions represent only the left half of the bridge.

The other half is made of concrete work again, which, important enough, contains every material relevant to reusage and big efforts of a user's investment.

4 Glue

As described in section ??, OOVs and the system they represent may be linked together in three ways: inheritance relations, dynamic inheritance relations and encapsulation relations.

There are two types of support for these relations necessary:

Design Level Support The first type of support is on the desgin level. That means providing tools for object oriented design which support OOVs and allow to specify the connection as easy as possible. Design level support is not described in this document.

Implementation Level Support The second type of support is to implement the desired functionality of the relations. This support may be considered as the *glue* between different OOVs and system parts.

4.1 Implementation Level Support

Support cannot be given in a general and complete manner, since encapsulation relations may be in fact anything. Therefore complete support means introducing a programming language, fortunately they do already exist.

But support can be provided for several subproblems, which saves work at least in some, if not in many cases.

Implementation level support is given by

The CyberSpace Object Architecture COA provides means to link and manage aspect dependent views on a per-object basis. COA objects provide mechanisms to access machine or human suited interfaces to them under given aspects and to manage these resulting interfaces. For details see [?].

Dynamic Object Models To support the generation and usage of OOVs at runtime, mechanisms for handling dynamic object models are needed. Clearly, this kind of support is not necessary for interpreted languages.

Integration, Synchronisation, Global Aspects To support various tasks like integration, synchronisation and other global aspects an instance is necessary which provides such services and works in conjunction with the respective (COA) objects. This instance may be an object representing the view, analogous to the COA `Shell` for a single object.

Other services may be of interest, which will be identified if there is some experience with OOVs. The classes `View` and `Shell` may serve as a useful platform for such services.

4.2 Classes

Next to the classes of COA ([?]), which provide a main part of support, some further classes shall be mentioned in the context of OOVs.

First, the classes `Class` and `Object` are used to support dynamic object models, if necessary. This is a task not only of interest in conjunction with OOVs and not explained in this document.

Second, the class `View` may serve for two purposes. There are two kinds of `Views`.

4.2.1 Class `ObjectView`

`ObjectViews` are similar to `Shells` of COA objects. They contain a group of objects composing a concrete view.

`ObjectViews` support various tasks like integration, synchronisation and other global aspects.

A good example of applying `ObjectViews` is view persistance. `ObjectViews` allow to store and retrieve complicated views and to move them around.

4.2.2 Class `ClassView`

In contrast to `ObjectViews` a `ClassView` contains a class model based on which several `ObjectViews` may be created.

Thus `ClassViews` are a kind of `Aspects`: they serve as a filter, which can be used as a parameter when creating an `ObjectView`.

`ClassViews` are more specific than `Aspects`. `Aspects` are typically an abstract description of points of view, while a `ClassView` is very concrete. It describes the exact form of the result the requester wants to have in terms of classes and their relations.

5 Taking Benefit

The concept of Object Oriented Views and their support on the design and implementation level has many advantages.

Small and Surveyable Class Models
can be created with OOVs without paying the price of uncomplete systems. There is no need for one big hierarchical class model describing the whole system. OOVs provide a mean to divide a system in several smaller class models. This is more suitable to describe complex and open systems.

Subject Orientation
is a major design principles made available by OOVs. This allows to concentrate on a specific user's needs, which gives the freedom to ignore uninteresting aspects and to use subjective ones.

Subsystem Encapsulation
Since OOVs provide a well defined boundary between subsystems and their environment, they serve as a valuable base for porting, adapting or maintaining them.

Leadership of Users
Using OOVs the users can get back the control over defining their needs: it is not longer necessary that a service provider defines what the state of the art is and the user has to adopt his system; instead the user defines what he needs and the service provider has to fulfill these needs. The roles change.

Saving Investments
Since OOVs primarily describe a specific users needs, they can be reused, even if the overall system changes. This is achieved since there is no integration of OOVs into one model and therefore the user specific information is preserved.

6 Conclusion

Object Oriented Views, providing small surveyable class models, may serve to meet the users needs of viewing and manipulating his environment.

They support the design principle of subject orientation and provide means for porting and reusing subsystems, thus helping to save taken investments.

OOVs are complemented by COA objects, together they provide a flexible and powerful base for building modern software systems.