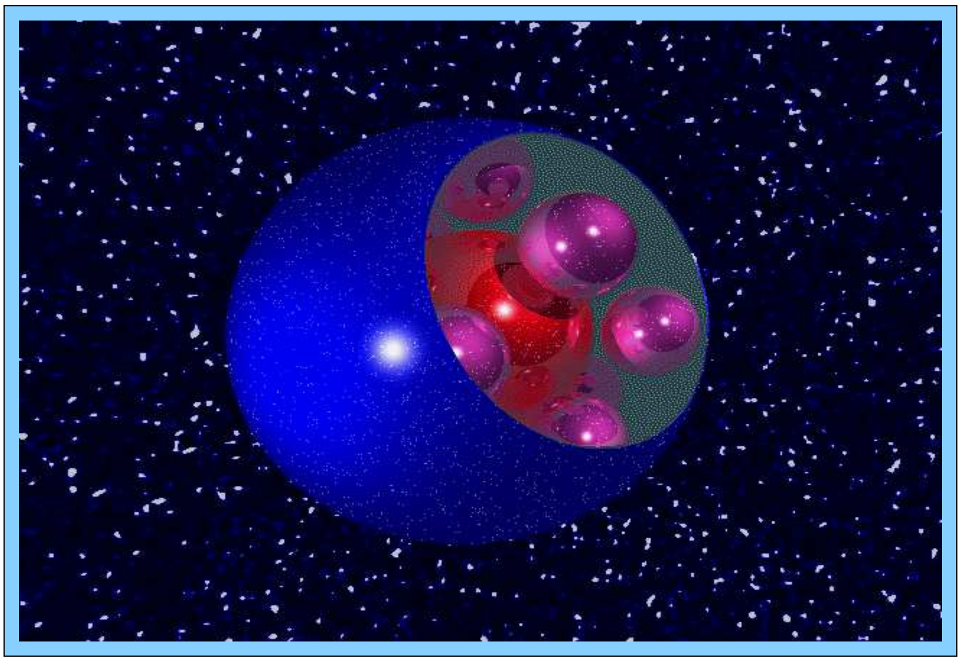
CAP

The CyberSpace Architecture Project

The CyberSpace Object Architecture

Building adaptable and rich equipped objects

*This paper describes the CyberSpace Object Architecture (*COA*), which describes a model for objects designed to meet the requirements given by todays complex and open systems and environments. The model is based on the object model of current object oriented techniques used in design and programming. It does not only extend this model, thereby introducing degrees of freedom between design and implementation, but also fits seemlessly into it, making integration of* COA *objects easy.*
The paper also shows the relation between COA*, the CyberSpace Foundation (*CSF*) and object oriented views (*OOV*) and the synergetic effects achieved.*



This picture illustrates the architecture of a COA *object. It's* `Shell` *(blue) encapsulates the* `Kernel` *(red) and the* `Accessory`*s (violet) surrounding the* `Kernel`*.*

1 Introduction

Why introducing another object model?

The COA object model is not a new model, but a straightforward extension and improvement of the common object model. It is based on and made of ordinary objects, and COA objects just look like these. COA is an object architecture, providing services and a framework for building sophisticated objects.

COA objects are dseigned to meet the requirements given by todays complex and open systems. The concept of COA is to focus from the start on openness. Objects placed in class hierarchies and concrete systems should not be bound there, as the hierarchies and the systems may change while the objects shall persist.

If system requirements cause too much tension on an object, the results of modeling it will most likely be unsatisfying, leading to either overloaded or uncomplete models.

COA provides means to relax this tension, without disregarding the various requirements. The equipment with rich information in a highly generic fashion enables COA objects to react flexible and adaptable to their environment.

2 Overview

Section ?? (INTRODUCTION TO COA) starts with a short recall of some properties of classes and their instances (section ??, INHERITANCE AND ASSOCIATIONS). Then, the COA object model is presented (section ??, THE MODEL).

Next, the different components are presented (section ??, THE COMPONENTS)): THE `Kernel` (section ??), THE `Accessory`S (section ??) and THE `Shell` (section ??).

The `Accessory` section contains examples of `Accessory`s together with an introduction to INHERITANCE VARIATIONS (section ??) and DIALOG ACCESSORYS (section ??), while the section about the `Shell` describes some basic services like `Accessory` MANAGEMENT (section ??) and COMPONENT ACCESS AND CONVERSION (section ??). An important service, the DYNAMIC CONVERSION (section ??) together with ASPECT TRANSFORMATIONS (section ??), is presented in section ?? (CONVERSION).

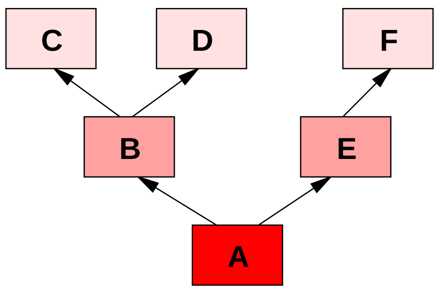
The following section (??), IMPLEMENTATION, is denoted to design issues: THE CLASSES (section ??) CLASS `Kernel` (section ??), CLASS `Accessory` (section ??), CLASS `Shell` (section ??) and CLASS `Aspect` (section ??) are described.

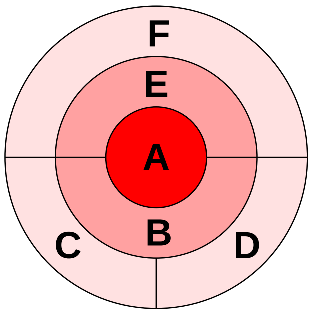
Finally, after mentioning SYNERGY EFFECTS (section ??) with the *CyberSpace Foundation* (CSF) in section ?? (COA AND CSF) and *Object Oriented Views* (OOV) in section ?? (COA AND OOV), some DRAWBACKS (section ??) should not be kept secret. But the following section ?? (TAKING BENEFIT) shows how to take benefit, and a CONCLUSION (section ??) is also given.

3 Introduction to COA

To understand the purpose of COA, some properties of objects and their relations have to be recalled here. The following section gives a short summary of these properties. Then, in the next section, the model itself is presented.

3.1 Inheritance and Associations





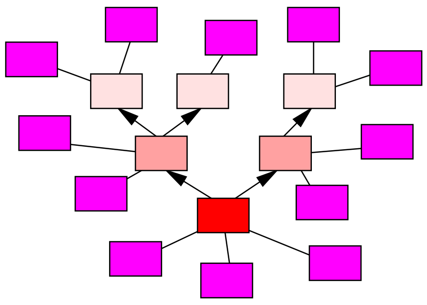
Class diagram (left side) and instance (right side) of a class A.

The figure on the left side shows a small class diagram of a class A and it's bases B – F.

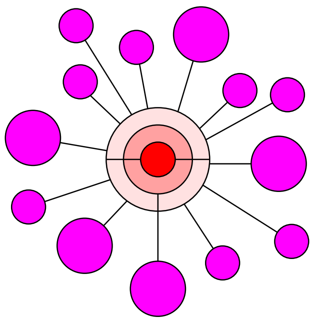
The right side shows an instance of this class, which is composed of the corresponding parts of each of these classes A through F.

Typically objects are related to various other objects, let's assume this is the case for our A, too. Such relations can be modelled as "associations" between classes.

The following figure shows on the left side the class diagram extended with some associated classes. The right figure shows A's instance completed by instances of the new classes and the respective links.



Class diagram with inheritance (arrows) and asscociation relations (lines).



Instance of the class with links (lines) and linked instances.

While there is much support for inheritance relations in object oriented programming languages (OOPL's), there is little or none for associations.

Supporting associations in general is not easy. Inheritance relations have a precise meaning, while an association relation only means "associated somehow". This is, of course, not a precise meaning.

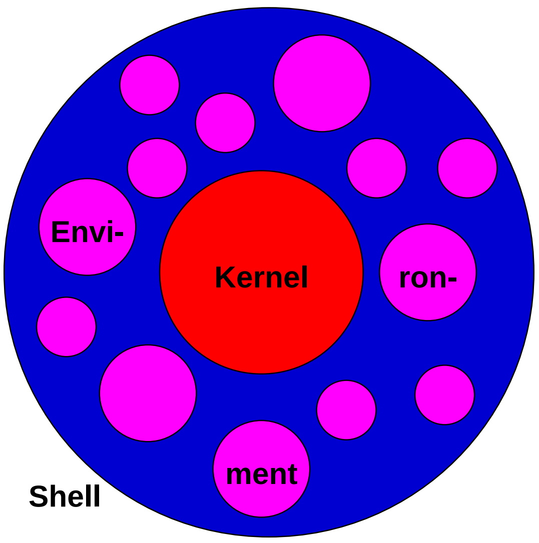
While it is difficult to support associations in general, there are subsets of associations for which proper support is possible. Support can be made in form of class libraries, CASE tools and code generators.

An important subset of associations supportable are those which are similar to "entity-relationship" relations in database applications. Those are the ones which are "first-class" relations in a good design since they are on the application domains level of abstraction.

Another class of associations are those which link an object to another object, which logically belongs very strong to the first one and serves for a special purpose as a kind of accessory to it.

This second class of associations and accessories is what COA deals with.

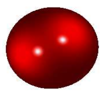
3.2 The Model



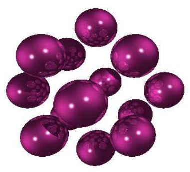
The Shell/Kernel/Environment Model.

The purpose of COA is to manage and support dynamic accessories of objects. It describes objects with dynamically created and managed components and how to access these components under various aspects.

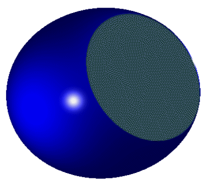
The COA object model describes objects as being composed of three components: *Shell, Kernel and Environment.*



The **Kernel** mostly resembles what a traditional class instance is, but is more pure in that it is restricted to the object essentials: *persistance relevant data* to ensure the objects identity as well as a *primitve but complete set of methods* to allow manipulation. Methods and data are as far as possible *subject independent.* The `Kernel` represents the *identity* of the object.



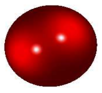
The **Environment** is composed of **Accessories**. This includes all kinds of objects which are related very close to the `Kernel` and do not have their own identity on the same level of abstraction. They are not statically bound to the `Kernel`, but form a *dynamic* and therefore *adaptable and felixble completion* of it. This does *not include* objects which are *part-of* the central object on an abstract design level. `Accessorys` are often *aspect dependend.*



The **Shell** is a hull around the `Kernel` and the `Accessorys`. It *manages and integrates* the inner components and provides *sophisticated access* to them under various aspects. It serves as a uniform interface to different kinds of objects, extending the functionalities provided by "the" root object found in many multi purpose class hierarchies.

4 The Components

4.1 The Kernel



The Kernel

- is a kind of pure traditional object, focussing on the essentials that make up the objects *identity.*
- contains *persistance relevant data*, i.e. data absolute necessary to be preserved.
- methods are *primitve*, but nevertheless *complete* in that they allow all necessary manipulations of the object's data.

No comfortable functions or combined methods are contained, no debugging or testing aids, no "print-me" or "store-me" methods, no statistic or otherwise administrative data, neither are Kernels derived from a class providing all these features, thus implicitly integrating them.

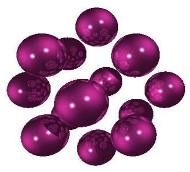
All these elaborated features are stripped from the Kernel and transferred into Accessorys. Some of these Accessorys will therefore need intensive access to the Kernels (private) data and have to be modeled as "friends".

This is **not** a violation of object oriented principles, as these classes are considered as a *dynamic extension* of the Kernel class with no own identity and no further purpose than this extension. The "encapsulation boundary" of the object is extended dynamically to these extensions.

From a logical point of view, these dynamic extensions are associated with the Kernel with special kinds of *is-a* relations. Examples and further explanations are given in the next chapter.

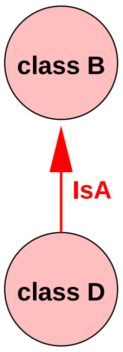
The separation of these parts from the Kernel allows the creation of objects which are more flexible, more oeconomic and performant. A clearer design can be achieved using the additional structure introduced by this separation.

4.2 The Accessorys



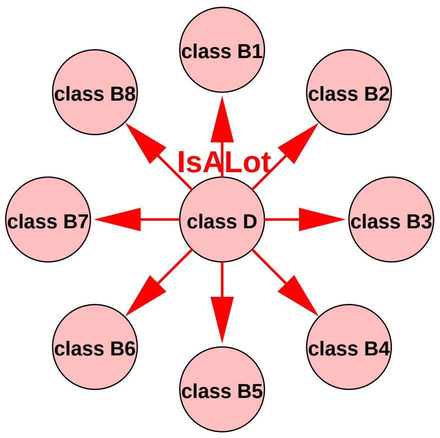
Currently the `Accessorys` can be devided in two categories. The first category contains objects whose classes are related in a *is-a* kind to the `Kernels` class. The second category contains classes like the *dialog classes* from the CSF, `Presentation` and `Manipulator`. These two categroies are not necessarily all nor do they assert to be complete.

4.2.1 Inheritance Variations



The classical inheritance relation says that all instances of a derived class D belong also to the base class B: a D *is a* B, with all consequences. This allows classifying common object properties by building base classes.
Objects may belong to more than one class. This is called multiple inheritance. Examining them exhaustively leads to the discovery that they may belong to really many classes.

Too many to be modeled, otherwise the model is neither surveyable nor maintainable.
A solution to this problem is to focus on the essential properties with respect to the application to build. The model has to be restricted to these properties. This approach has one great disadvantage: objects modeled under a certain aspect are not easy to reuse in or to port to different environments, which is an important need.

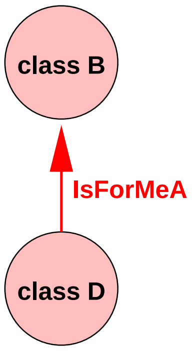


But there is another solution: examining the different inheritance relations, an important observation can be made:

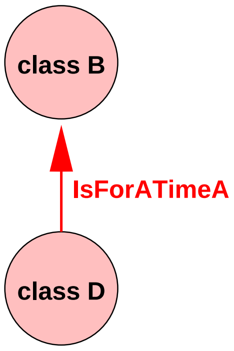
not all base classes are needed all the time and by everyone.

On the contrary: at a given point of time a user of a class only needs a very small subset of all base classes.

These *variations of inheritance* are shown in the following figure.



Variations of Inheritance
Two may-be-a relations are shown. The first one is a **is-for-me-a** *relation: D belongs only for me to B, not for everybody. The other one is a* **is-for-a-time-a** *relation, which can have two forms, depending on which class is temporary: a temporary B provides a temporarily needed access type to D, a* **view***; a temporary D represents a* **role** *an object of B can take on.*



Thus, by considering only certain well and explicit defined subsets of base classes and managing them dynamically, all base class requirements can be satisfied in a surveyable manner.

In COA, these temporary or aspect dependend classes are modeled as `Accessorys`. They are created on demand with the neccessary access privileges to the `Kernel`.

It should be noted, that these dynamic base or derived classes do not represent objects with their own identity.

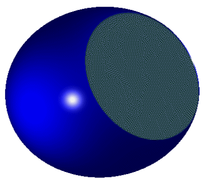
4.2.2 Dialog `Accessorys`

In OOD models or OO programs a special category of objects can be found, which are not first-class candidates for the design but are nevertheless somehow necessary. These objects often deal with the presentation or manipulation of other objects.

Because these objects have a more technical character, but are too important to be hidden on a lower level, it suggests itself to model them as `Accessorys`. In CAP objects dealing with presentation and manipulation are modeled within the CSF as the *dialog classes* `Presentation` and `Manipulator`.

For further details see the CAP document on dialog classes.

4.3 The `Shell`



Finally, the COA object is completed with a `Shell`, integrating the `Kernel` and the `Accessorys` and providing an intelligent interface to these components.

The `Shell` can be compared to "root" classes (named like "The Object") found in many multi purpose class libraries, which serve similar functionalities.

Such base classes and the respective relations are "correct", as each object *is-in-fact-an* (OOPL) object, but clearly on a very different level of abstraction than the application domain, where OOPL-objects are rarely found.

Independently of the correctness of this "one base class approach" and it's usefulness, it loads the burdon of many methods, data and features to each object. This load can be reduced by providing the services dynamically.

4.3.1 `Accessory` Management

The first task the `Shell` performs is to manage the `Accessory`s internally. This includes services like maintaining lists of created `Accessory`s, retrieving them if they are needed again, cooparate with a garbage collector and so on.

These management tasks are specific to the kind of `Accessory`s available for an object.

4.3.2 Component Access and Conversion

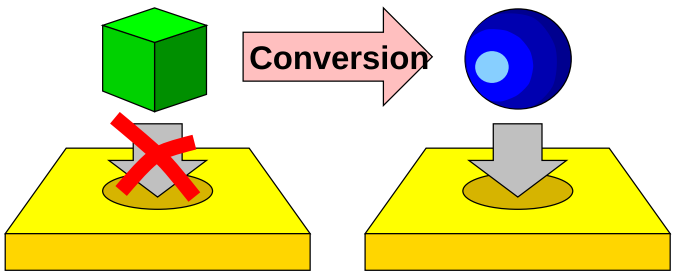
The other task, which is conceptually more interesting, is to provide access to the components in an intelligent manner. That means having a standardised way of access, which can, first of all, be used very easy and, if necessary, extended to a very sophisticated level of communication to specify what is needed.

This prepares objects for the usage in open environments. Two kinds of access can be distinguished.

- The first one is access to `Accessory`s, where the original object is still needed. An example are the dialog classes: access to a `Presentation` of an object, while, clearly, the original object can still be needed.
- The second kind of access is to dynamic base classes, as metioned in section ??. This kind of access resembles a conversion, where another view of the object is needed and the old one is no longer used.

An example for the first kind of access is given in section ??. The conversion-like access is described in more detail in the following section.

4.3.2.1 Conversion



Conversion is used to make the form of an object appropriate.

Consider the C++features related to conversion, ignoring for this time the difference between conversion and cast:

- First, as each derived object of a class D *is-an* object of the base B, references to D's are converted automatically to references to B's where needed.
- Classes can provide explicit conversion-functions, which are used automatically in some situations.
- The same holds for conversion by constructor, where a new object is created based on a given one.
- If this is not sufficient, the programmer can use explicit conversion, if he knows what he is doing.
- `dynamic_cast< >`[1] can be used to convert to a derived class with a safety check.

All these features are often used to convert a given object into a form which is more appropriate for the task to be performed with it.

Though these features are evolving and getting more flexible, some common diffculties can be observed: each step of a conversion has to be specified precisely on a low level, probably involving some checks, and the link between source and target of the conversion has to be managed.

An approach to solve these diffculties is *dynamic conversion.*

[1]This feature is not yet available but shall be included in future versions of C++

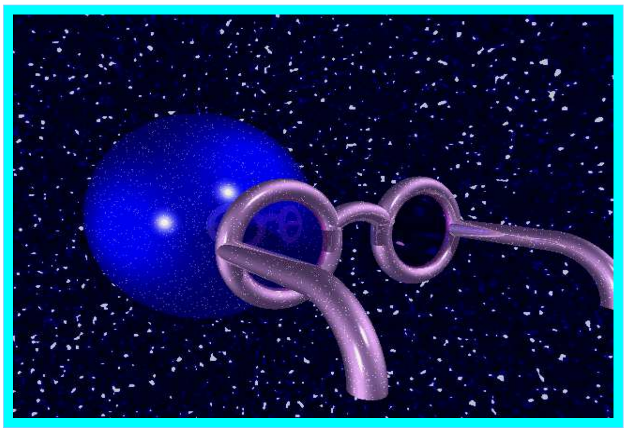
4.3.2.1.1 Dynamic Conversion

Dynamic conversion is a mechanism unifying the above mechanisms and providing a flexible conversion interface, which allows to specify the target on a more declarative level, relying on the power of object oriented abstractions.

The idea is simply to have an interface to an object which allows to specify a target class and some additional information to perform the conversion. The kind of conversion internally performed is completely hidden from the requester, and he doesn't have to care about destruction and similar aspects concerning the source object.

The `Shell` is a good anchor for this conversion mechanism, since it is the first address of a COA object to request a view from.

4.3.2.1.2 Aspect Transformation



An object can be viewed under different aspects, leading to different views of this object.

To charactarize the points of view one can take on and to capture this information the concept of an *aspect* is introduced. An `Aspects` is a (technical) object, which carrys information describing points of view on a more or less abstract level.

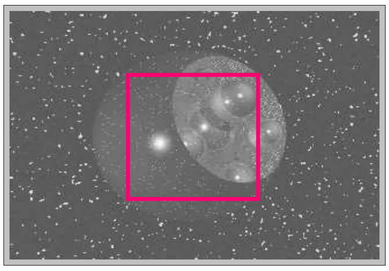
`Aspects` are used as "parameter objects" for the dynamic conversion mechanism previously described.

As an example the dialog classes can serve again. Consider a request to an object to provide a graphical presentation of itself. There are many ways of giving such a graphical presentation: a technician may want to have a CAD figure showing the components and their states, while the manager would prefer a graphic showing the costs and the amount of work saved. Obviously there are two aspects: a technical and a financial one.

5 Implementation

5.1 The Classes

5.1.1 Class `Kernel`



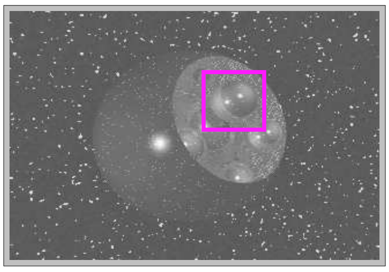
class Kernel

The class `Kernel` exists mainly to capture the *concept* of a kernel. There are few or no functionalities associated with this class, since `Kernel`s are reduced to the essentials of an object, all more general services are moved to the `Accessory`s.

One service, a `Kernel` can provide, is to maintain a link to the associated `Shell` object, which is able to satisfy requests of many kinds.

It is possible, that, e.g. for efficiency reasons, the kernel of a COA object is not modeled as an instance of class `Kernel`, but as an object completely determined by the application domains requirements. Then, it is of course not possible to access the `Shell` from the kernel, but the other direction is still possible.

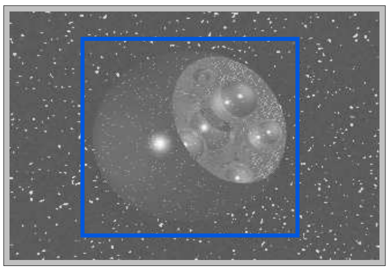
5.1.2 Class `Accessory`



class Accessory

The same as for the `Kernel` holds for `Accessory`s. If the various accessories are modeled as instances of class `Accessory`, it is mostly for access purposes. Of course, one can e.g. put all `Accessory`s and the `Kernel` in a linked list to retrieve them. But these are implementation issues. Most important on the conceptual level is access to the `Shell`, since it is the object providing all the further services.

5.1.3 Class `Shell`



class Shell

Since the `Kernel` and the `Accessorys` have nearly nothing to do, someone else will: the `Shell`.

The `Shell` serves as a base class for various specialised kinds of shells and other services. It's main task is to accept requests of many kinds and to route them to some instance in the COA object which is able to satisfy the request.

Currently there are the following services defined. The list is not complete, it only contains sufficiently abstract services discovered so far.

Dynamic Converter
: This service performs *dynamic conversions* as described in section **??**. It accepts conversion requests and returns the respective results.

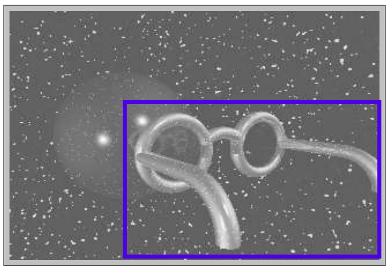
Dialog Service
: The dialog service resembles the dynamic conversion service, since also requested classes are retrieved. But dialog classes (`Presentation` and `Manipulator`) are intended to be used by a human, while ”normal” classes are made for algorithms.

Structure Service
: Many objects have an internal structure. To access their components, it may be useful to have a general mechanism available. It is at least useful for humans, to ”have a look” into the object.

Adaptation Service
: This service allows objects to adapt themselves to a given environment. This is a feature which is most useful in conjunction with the *space classes* of the CSF, see section **??**.

All these services do not necessarily have to be linked to the respective object. It is possible, that an object does not provide them. The COA structure defines a base to link such services to, but it does not require each object to provide them all. Most likely these services will be defined once and used by many objects, which specialise the features if necessary.

5.1.4 Class Aspect



class Aspect

Aspects are an abstract description of the relation subject–object (viewer–viewed thing). They allow to capture the relevant information necessary to create a view.

Aspects are related to the semantics of a view, therefore it is not easy to classify them. The full comfort one may want to have will need more research and development efforts, but as a first step they provide an anchor to connect such information to. Some common aspects will likely be not too difficult to define.

6 Synergy Effects

COA is designed to supply objects with flexibility, adaptivity and dynamics. These properties are useful by themselves. Clearly, an environment that can make use of them will intensify the benefits.

The *CyberSpace Foundation Classes* (CSF) and *Object Oriented Views* (OOV) are designed to work with COA and partly rely on COA objects.

Some examples of the interaction between these components of CAP will be given here.

6.1 COA and CSF

The first example concerns the interaction between COA and CSF. More precisely, the *space classes* and the *dialog classes* are mentioned.

6.1.1 Space Classes

Since COA objects are able to change their appearance and they are prepared to be viewed under different aspects, they are able to travel through the CyberSpace and adopt to their respective environment. This can be done automatically using `Areas` and `Gates`.

For more information on these classes see [?].

6.1.2 Dialog Classes

An important type of `Accessory` are the dialog classes `Presentation` and `Manipulator`. These classes represent the interface between humans and an object.

A COA object provides the necessary information in a generic fashion, to allow the construction of nearly arbitrary instantiations of such interfaces.

This includes the provision of components or the generation of complete GUI's, documents and reports, integration of information in hierarchic information systems, simple text interfaces etc.

For more information on these classes see [?].

6.2 COA and OOV

The second example illustrates the interplay between COA and OOV.

Object Oriented Views allow the creation of *subject oriented* class models, which represent a "personal" view on a complex system. The system needs not to be described by only one big class model, representing a compromise of the different views.

To preserve object identity and to provide different interfaces to the objects as needed by the different views, COA objects are useful to provide the necessary flexibility. COA objects thus act as the "glue" between the different views.

For details on OOVs see [?].

7 Drawbacks

A lot of the features presented in this document are of course features one can imagine to be integrated in future OOPL's. Since the approach choosen here is to use a class library, the integration with the programming language used is not given to an extent as it is desirable.

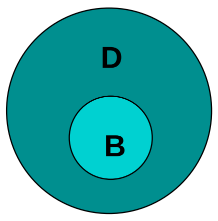
An approach to solve this problem is to use code generators and design tools. There are lots of tasks a code generator can do, not only improving the integration of these features.

Ideally, it creates, given on a short description of the `Kernel`, a complete COA object with a rich set of default functionalities. These can than be modified in more detail, if necessary.

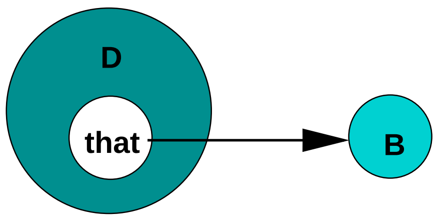
A special and more fundamental problem should be metioned here. This problem is related to the extraction of features from the `Kernel` object into dynamic accessory objects.

These accessory objects have to be bound to the kernel somehow. Ideally, as shown in the following figure, this would be performed using a "that-pointer", linking the two objects together as a form of realising an inheritance relationship.

The term "that-pointer" is choosen to resemble the "this-pointer" in C++ objects.



An instance of a class D with a base class B.



The same object with a dynamic bound instance of B.

The problem is, that between objects linked this way there is primarily no longer any support of object oriented features like dynamic binding (polymorphism), being in a class's scope or privileged access. [2]

As a workaround for this a precompiler can serve. Alternatively, a code generator can be able to resolve the relevant scoping problems. The most difficult task is to resolve dynamic binding between the distinct objects.

[2]Providing this form of linkage is not an easy task, neither on the semantic level of the programming language nor on the implementaion level for compiler builders.

8 Taking Benefit

The COA architecture is designed to introduce flexibility and dynamics into objects. This can be used with profit for several purposes:

Saving Work
COA objects can save work, since a lot of effort for managing and equipping objects has to be done only once within this architecture. The generation of the necessary equipment can be automated to some extend.

Resolving Design Conflicts
Providing more and especially dynamic instruments for structuring objects COA can help to resolve design conflicts without paying the price of managing encapsulated behaviour. Systems can be broken down in even smaller pieces, one can focus on and survey.

Adaptive Objects
COA objects are provided with abilities for adapting themselves to different environments, since the information is available in a generic fashion. Also they can adopt to different requirements in the same environment, depending on the availability of services and resources.

Oeconomic Resource Management
Not every feature a COA object may need has to be available at any time but can be created dynamically on demand. Used for many objects in a system, this can save noticeable amounts of system resources.

Supporting Portability
Since COA objects are designed to meet new requirements, they are not bound as strong to a given class model as ordinary objects are. The possibility to embed them in different class models is part of the concept, and not a problem arising when protability is requested afterwards.

Cooperativity
COA objects do not only allow the existence of other components, such as class libraries, it is a central goal of COA to support the coexistence with them. Since there are primarily no requirements on `Kernels`, each instance of any arbitrary class can serve as a `Kernel`.

9 Conclusion

The CyberSpace Object Architecture, based on standard objects and being a straightforward extension of them, can help to meet the requirements of todays complex and open systems.

COA objects are flexible, adaptive, and prepared for porting. Using the architecture can save work, resolv design conflicts and allow oeconomic management of resources.