Complex Associations: Abstractions in Object–Oriented Modeling *

Bent Bruun Kristensen

Aalborg University
Institute for Electronic Systems
Fredrik Bajers Vej 7, DK-9220 Aalborg Ø, Denmark
e-mail: bbkristensen@iesd.auc.dk

Abstract

Objects model phenomena and a phenomenon is usually a component. Information characterizing a component is encapsulated and accessible only by its methods. The *relations* between components are modeled explicitly by means of associations or references. A relation is also a phenomenon and objects can model this type of phenomena too. Components are usually related conceptually in diverse and subtle ways: Some relations are implicitly given and some are local to other more basic relations. Such kinds of relations are important for understanding the organization and cooperation of objects and may be supported in object–oriented analysis, design, and programming: An *implicit association* describes a relation between an object and objects *local* to this *enclosing* object, and a *complex association* describes an explicit relation between *local* objects in different enclosing objects. Such associations are described by classes and the objects have the usual properties including methods and attributes.

1 Introduction

One of the strengths of object–oriented modeling – the isolation and encapsulation of information as objects – is also one of its major problems. Information that characterizes an object is hidden inside the object and is usually accessible only by methods. In this context relations between objects are explicitly modeled during design by "associations" and references between objects. However, other types of phenomena exist, such as activities and relations, which have a rich and natural conceptual identity of their own outside of any individual component. These are in conflict with the encapsulation in objects.

In this paper we argue that we need to pay much more attention to implicit relations between objects, and to information that exists between and external to objects. The attention must be reflected in the methodologies and the description mechanisms for object–oriented analysis, design and programming. As a step in that direction we introduce language mechanisms to support object–oriented modeling of *implicit* and *complex* associations:

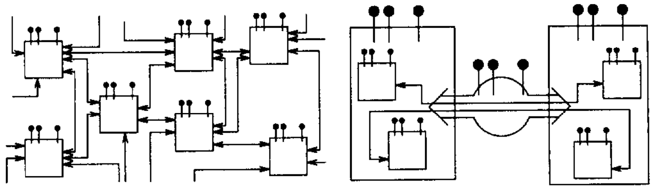
- Implicit associations are described by means of enclosing classes. Objects of such classes have the usual properties such as methods, attributes, etc.
- Complex associations are described by means of association classes. These associations may

---

[*] This research was supported in part by the Danish Natural Science Research Council, No. 11-0542-1, while the author was on leave at Department of Computer Science, Monash University, Australia.

OOPSLA 94- 10/94 Portland, Oregon USA

also be enclosing classes. Association objects have the usual properties of objects.



a) Flat Model: Objects and Associations

b) Structured Model: Complex Associations

Figure 1: Illustration of Problem and Solution

To give a flavor of the approach Figure 1a is a schematic illustration of the usual flat model with classes/objects and a lot of associations, whereas figure 1b illustrates a nested, structured model with complex associations.

In section 2 we discuss the use of implicit and complex associations as they apply to the the modeling process. We focus on the analysis phase where we are trying to understand and describe some part of the world. We present a typical model from the literature, which is hard to understand, and then illustrate alternative models by introducing implicit and complex associations. In section 3 we propose a set of related programming language mechanisms to support the processes and models illustrated in the previous section. In section 4 we discuss related work. In section 5 we review an experimental project focused on the design, implementation and application of the ideas in this paper, and summarize the experience from the project. In section 6 we summarize the proposals and results of the paper. A summary of the language mechanisms, additional shorthand notations, as well as various specialized forms of the mechanisms are given in the appendix.

2 Modeling Complex Structures

In the process of modeling some part of the world in an object–oriented fashion the focus is on identifying concepts and their mutual relations and then describing these by means of classes and associations between them. Using existing methods and notation we usually describe all the classes and the corresponding associations between them in one, flat model, despite the fact that these are typically at different levels of detail. Consequently the description often appears confusing and disorganized. The problem is that this kind of description does not reflect the way that we think about and understand such complex systems.

"Automated Teller Machine". As an example consider the "Automated Teller Machine Example": The purpose is to design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATM's) to be shared by a consortium of banks. The object diagram presented in [Rumbaugh et al. 91] regarding this example includes classes modeling `Consortium`, `Bank`, `Account`, `Customer`, `Computer`'s, `Cashier`, `ATM`, `Transaction`'s among others. Examples of the associations between these are `Consists-Of`, `Has`, `Entered-On`: a `Consortium` `Consists-Of` some `Bank`'s, a `Customer` `Has` an `Account`, a `Transaction` is `Entered-On` an `ATM`. The diagram is given in figure 2.

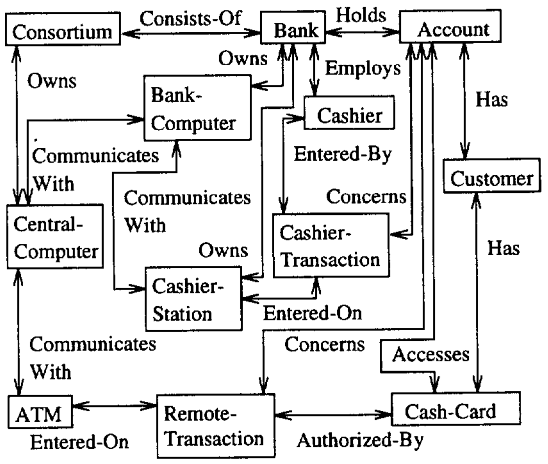


Figure 2: ATM–System from [Rumbaugh et al. 91]

We claim that the classes are related in more complex structures than such existing object–oriented analysis and design methodologies support. Part of the reason is the lack of appropriate language mechanisms for expressing (and thus guiding in the description of) these complex structures and relations. Our approach to supporting the structuring of the modeling process in a better way is to offer more powerful description mechanisms.

Banking Example: Top Level. We assume that figure 2 is a valid model, given some perspective on this problem. However, the model is not comprehensible and does not correspond to our usual way of thinking, namely: At the top level we think of a relation like banking as an association, say Banking, between some banking company (the Consortium in the example) and some general Customer. The Consortium and Customer are the domains of the Banking association. This top level is illustrated in figure 3.

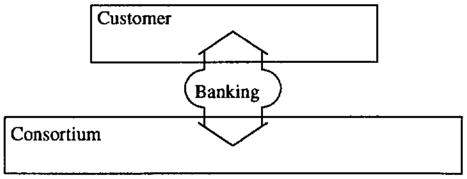


Figure 3: Banking: Customer–Consortium

Banking Example: Next Level. At the next level (Figure 4) we think of the bank branches of the Consortium and the ATM's situated in various places: We see the concept Consortium organized as a number of components according to the concepts Bank and ATM (and in the example also Centralized-Computer). Similarly, the Customer concept may be organized into components, especially when we imagine households or companies as examples of different kinds of more complex customers. We choose to organize Customer in Cash-Card and Account. (No Person is included in the example.) The Consortium is called an enclosing component. Therefore, on the one hand, Consortium is itself a component and will be modeled as such. On the other hand, it consists of a number of components local to it (similarly for Customer). [1] The associations between the components of this next level can exist only because of the association between the enclosing components at the top level, and they are local to the association on the top level. In the example the Account is associated with Bank by Holds and with ATM by Concerns. The Cash-Card is associated with ATM by Authorized-By. Therefore associations such as Holds, Concerns and Authorized-By are local to the association Banking. The domains of these local associations are domains local to the domains of Banking.

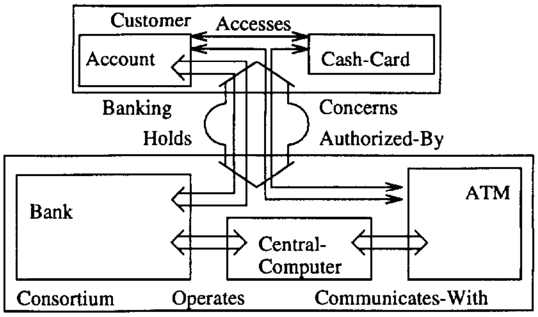


Figure 4: Banking: Customer–Consortium, Cont.

The enclosing Consortium has the components Bank, ATM and Centralized-Computer local to it. Therefore our model has an implicit association between the enclosing component and each of its components. Regarding Consortium, we find that Owns for Central-Computer, and Consist-Of's for Bank's are explicitly given in [Rumbaugh et al. 91], whereas the one for ATM's is not. Similarly, in our model we have implicit associations between each of the associations Holds, Concerns and Authorized-By being local to the association Banking. Between enclosing components we have the explicit association between Consortium and Customer, namely Banking and its local associations. Inside an enclosing component we have explicit associations between the local components, such as Accesses

---

[1] We use the term *local to* informally with the meaning *belonging to* or *existing in*, in contrast to *part of*, with the meaning *some but not all of a thing.*

for `Account` and `Cash-Card`, and `Communicates-With` for `Central-Computer` and `ATM`, whereas an association like `Operates` for `Central-Computer` and `Bank` is added in our example.

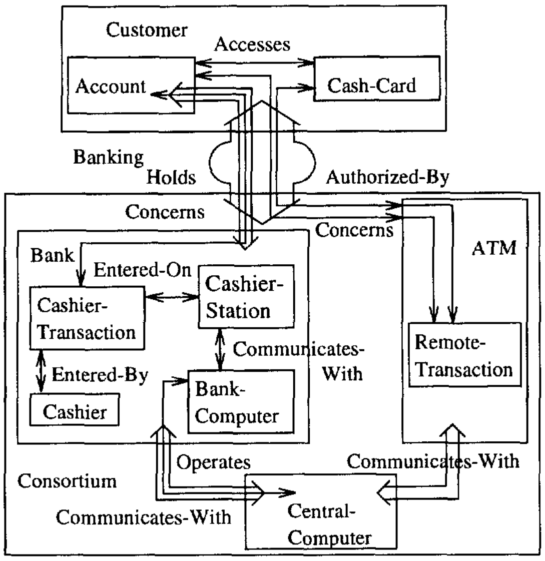


Figure 5: Banking: Customer–Consortium, Cont.

Banking Example: Final level. We illustrate the next level (Figure 5) for the `Consortium` only. Thinking of a branch, we find `Cashier` and `Cashier-Station`. Also we realize that we have a `Cashier-Transaction` and we may even have a `Bank-Computer`. Therefore `Bank` is itself an enclosing component with these kinds of components. Similarly, `ATM` is an enclosing component with `Remote-Transaction`'s as local components.

With respect to associations, `Authorized-By` is an association between the components `Cash-Card` and `Remote-Transaction` (local to, and replacing `ATM` in this association). Similarly, with respect to `Concerns` between `Account` and `ATM`: These are examples of associations which are extended from the enclosing component to a local component. The `Holds` association between `Bank` and `Account` may be seen as a complex association, so that the association `Concerns` between `Account` and `Cashier-Transaction`'s is local to `Holds`. The domain `Cashier-Transaction` is local to the domain `Bank` but the domain `Account` is unchanged. The association `Operates` for `Central-Computer` and `Bank` is another example of a complex association: The association `Communicates-With` between `Central-Computer` and `Bank-Computer` is local to `Operates`. `Bank-Computer` is a domain local to `Bank`, whereas the domain `Central-Computer` is unchanged.

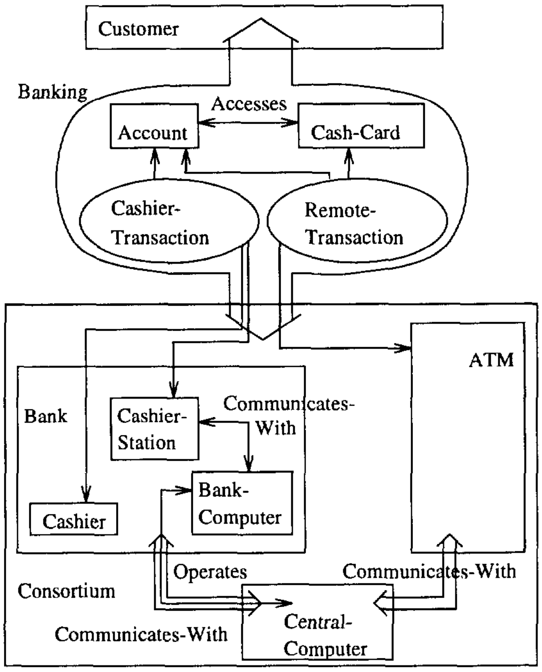


Figure 6: Banking: An Alternative Model

Banking: Alternative Model. Finally, we discuss an alternative model (Figure 6). Previously we chose to consider `Customer` as an enclosing component with `Account` and `Cash-Card` as local components. An alternative is to let `Account` and `Cash-Card` be local components of `Banking` and also to let `Banking` be an even more complex association, not only including local associations, but also local components. This seems natural in the example, but may be even more natural if `Customer` had been some company with various departments, local to the company. Therefore `Banking` is an association (with local components `Account` and `Cash-Card`) between `Consortium` and some kind of `Customer`, which is not specified any further. Similarly, we may let `Cashier-Transaction` (respectively `Remote-Transaction`) be the association itself – and not a component – between `Account` and `Cashier` and `Cashier-Station` (respectively between `ATM` and `Cash-Card` and `Account`). In this last case we may still consider `Account` and `Cash-Card` to be local components of `banking` so that `Cashier-Transaction` and `Remote-Transaction` are associations between components, which are local to either associations or components.

3 Language Mechanisms

Our main point is that the complex structures which appear in the previous section are not only a practical way to illustrate a model, but are also the way that we actually think about it: At the top level we have some rather complex `Banking` association and at a more detailed level this association is refined into several simpler associations between several simpler components. By distinguishing between the top level and the next level we have not just given alternative presentations of the same model. We propose that the concept *local to* is available through some supporting language mechanism. Moreover, the concept is available for both components and associations and at several levels. [2]

The description of a local component is meaningful only in connection with the description of the enclosing component, and the existence of a local component is dependent on the existence of an enclosing component. The description of a local association is meaningful only in connection with the description of an enclosing complex association, and the existence of a local association is dependent on the existence of an enclosing complex association.

We need language mechanisms to describe implicit and complex associations. For this purpose we introduce slightly modified general classes and objects with methods, attributes, etc.

The following is a schematic description of a general class: Methods, `M`, and references, `R`, are in class `C` and may be accessed by means of "dot"-notation, such as `aC.M` and `aC.R`, where `R` is a name of a reference to some object of class `C'`. [3] The class `C` and a graphical illustration of it are given in figure 7.

```
C: class
( ... M: method(...)
  ... R: reference C'
)
aC: object C
```

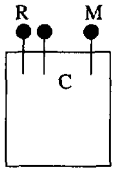


Figure 7: Notation and Illustration of a Class

In the schematic examples we shall use `c:(...)` (or `a`, `b`, `r`, and `z`) to denote methods, references, etc. We use the generic term "attribute" for any of these, and do not distinguish between them with respect to visibility from outside an object.

Basic Associations. Given two domains in the form of classes `A` and `B`, with attributes `a` and `b`, respectively, a *basic association* `R`, with attribute `r` (figure 8a), may be declared as follows:

```
R: --class [A,B] ( ... r:(...) ... )
A: class ( ... a:(...) ... )
B: class ( ... b:(...) ... )
```

An association class is indicated by the notation `--class`. The body of `R` is similar to a usual class descriptions and may include various other parts.

We shall include two extensions: `A` objects may play a role named `roleA` for `B` objects in relation to `R`, and `R` may be a one-to-many association, from one `A` to many `B`'s:

```
R: --class [roleA: A, roleB:* B] ( ... r:(...) ... )
```

---

[2] We use the term *local* to distinguish between the local objects and the part objects of some (whole) object: An (enclosing) class/object may have *local* classes/objects.

[3] We use `reference` in the declaration of names of local objects. Even though our objective is to introduce alternative abstraction mechanisms, we do not claim that primitive references can be replaced completely.

Given an `A` object `anA` and a `B` object `aB` the instantiation of the association object between these objects is denoted: `--object R (anA, aB)`. The selection of this specific `R` object is denoted: `R(anA, aB)`. Furthermore, `R(-,-)` denotes the set of association objects instantiated from `R`.

The language mechanisms are illustrated by the `Banking` example. The following description illustrates the `Banking` example at the top level (figure 9a):

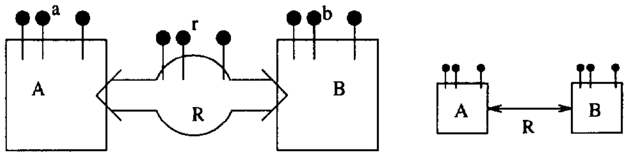
```
Customer: class
( ...
  Account: class ( ... List: method(...) ... )
  Cash-Card: class ( ... TheAccount.List ... )

  Accesses: --class
    [TheAccount: Account,
     TheCash-Card: Cash-Card] (...)

... TheConsortium ... )

Consortium: class ( ... TheCustomer ... )

Banking: --class
[TheCustomers:* Customer, TheConsortium: Consortium]
(...)
```

`Banking` and `Accesses` are basic associations (local to `Customer`). An association object between a `Customer` object, `BBKing`, and a `Consortium` object, `USavings`, may be instantiated as follows:

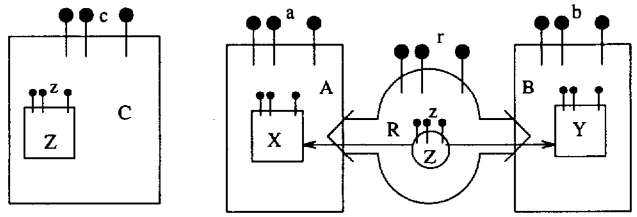
```
--object Banking (BBKing, USavings)
```

Basic Associations: Access. The language mechanisms introduced in this section support not only a description of the *static* relational structure developed in the previous section but also direct *dynamic* access to the objects in the complex structure. A summary of the notation is given in the appendix.

Dot–notation is available for accessing attributes of objects and association objects: The `r` attribute for the association object between the objects `anA` and `aB` is denoted as `R(anA, aB).r`. The `r` attribute for all objects in `R(-,-)` is denoted `R(-,-).r`. More generally, `R(-,-).{...}` means the execution of the action sequence "..." for all the objects in `R(-,-)`.



a) Basic Associations



b) Implicit Association

c) Complex Association

Figure 8: Illustrations of Associations

The `anA` object is denoted by `R(-,aB)` and the `a` attribute of `anA` is denoted by `R(-,aB).a`. Inside the object `aB` the object `anA` is denoted `R'roleA` only:

```
B: class ( ... R'roleA ... )
```

Similarly, `R'roleA.a` denotes the attribute `a`.

`R(anA,-)` denotes a set of associated `B` objects. `roleB(anA,-).b` is a multiple access of the `b` attribute of a set of `B` objects. Similarly, `R'roleB` used inside the `anA` object denotes a set of `B` objects.

The actual objects in the `R`–association object are accessed by means of `roleA` and `roleB`, as for example: [4]

```
R(-,-).{ ... roleA ... roleB ... }
```

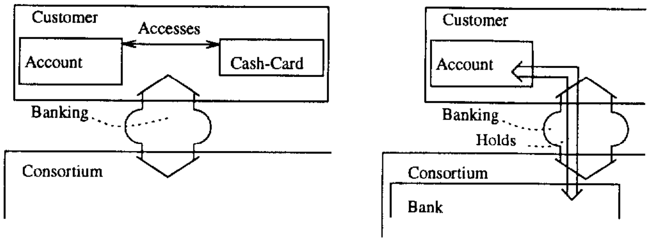
A special notation is available from inside an associated object such as `aB` for the access of `r`, namely `R'r`, which denotes the `r`'s of the set of association objects between `anA` and some `B` objects.

The description of the `Banking` association illustrates examples of access of objects and methods: From `Customer` the `Consortium` is accessed by

---

[4] The meaning of `someC.{...M...R...}` is: `someC` is an object or association object (or some set of these). "`...M...R...`" is some action sequence with denotations of some methods and references, respectively `M` and `R`. The action sequence is executed with the substitutions "`...someC.M...someC.R...`". If `someC` is a set of objects, the action sequence is executed for each object or association object in the set.

`TheConsortium` and from `Consortium` the `Customers`'s are accessed by `TheCustomer`. From `Cash-Card` the method `List` of the associated `Account`'s is accessed by `TheAccount.List`.



a) Banking Illustration

b) Banking Illustration

Figure 9: Illustration of Example

Local Classes and Objects. An enclosing class is a usual class except that local classes may be declared (syntactically) inside the enclosing class. An enclosing object may also have attributes. Objects of a local class are local to an enclosing object. A local object is dependent on the existence of the enclosing object.

The enclosing object `C` has the attribute `c:(...)`. The local class `Z` is nested inside enclosing class `C`, so that the objects of `Z` (for example an object referenced by `aZ`) are local to the enclosing `C` object, for example `aC` (figure 8b):

```
C: class
( ... c:(...) ...
  Z: class ( ... z:(...) ... )
  aZ: reference Z
...)
aC: object C
```

The class `Z` (together with class `C`) also introduces an *implicit association* between objects of `C`, such as `aC`, and its `Z` objects.

An `aC.Z` object is instantiated by `--object aC.Z`. The set of `aC.Z` objects local to `aC` are denoted [5] by `aC.Z(-)`.

In the `Banking` example the class `Customer` has the local classes `Account` and `Cash-Card` (figure 9a). The `List` method of the `Account`'s of a `Customer`, for example `BBKing`, is accessed by `BBKing.Account(-).List`.

Enclosing classes may be nested, i.e., they may have local objects at several levels. The `Consortium` has the local class `Bank`, which has the local classes `Cashier-Transaction` and `Cashier-Station` (figure 10a):

```
Consortium: class
( ...
  Bank: class
  ( ...
    Cashier-Transaction: class (...)
    Cashier-Station: class (...)
  ... )
... )
```

Complex Associations. An association class may also be an enclosing class with local association classes.

Class `A` has a nested class `X` and a local object `anX`. Similarly for `B`, `Y` and `aY`. Furthermore, `R` is an association between `A` and `B`. Nested in `R` we describe a *complex association*, `Z`, between `X` and `Y` (figure 8c):

```
R: --class [roleA: A, roleB: B]
( ... r:(...) ...
  Z: --class [roleX: A.X, roleY: B.Y]
  ( ... z:(...) ... )
... )

A: class ( ... X: class(...) ... anX: object X ... )
B: class ( ... Y: class(...) ... aY: object Y ... )
```

Given `A` and `B` objects and an association object of `R` between these, we can instantiate an association object of `Z` between the `anX` and `aY` objects from inside `R` by:

```
--object Z (roleA.anX, roleB.aY)
```

The `Banking` association has a local association `Holds`. The domains of `Banking` are `Customer` and `Consortium`. The domains of `Holds` are `Bank` (local to `Consortium`) and `Account` (local to `Customer`) (figure 9b):

```
Banking: --class [ ... ]
( ...
```

---

[5] We use the dot notation (for example `aC.Z`) to simplify the description, and we leave it open whether or not `Z` objects should be accessible from inside `C` objects only.

```
  Holds: --class
    [TheBank: Consortium.Bank,
     TheAccount:* Customer.Account]
  ( ... Expiration-Date: method(...) ... )
... )
```

Complex Associations: Access. In the association `Z` the objects of `X` and `Y` may have the roles `roleX` and `roleY`, respectively. For `X` and `Y` objects the association `Z` is available in exactly the same way as `R` is available for `A` and `B` objects. Inside `X` (respectively `Y`) the corresponding objects may be denoted by `Z'roleY` (respectively `Z'roleX`). Inside `R` the operations for `Z` and the notations `roleX`, `roleY`, etc., are directly accessible.

The set of associations between a `Bank` and an `Account` object is denoted by `Holds` and a method `Expiration-Date` for `Holds` is accessed for all elements in this set by `Holds.Expiration-Date`. To access the `List` method for all the `Account` objects (local to a `Customer` object) from inside a given `Bank` object (local to a `Consortium` object) we use the notation:

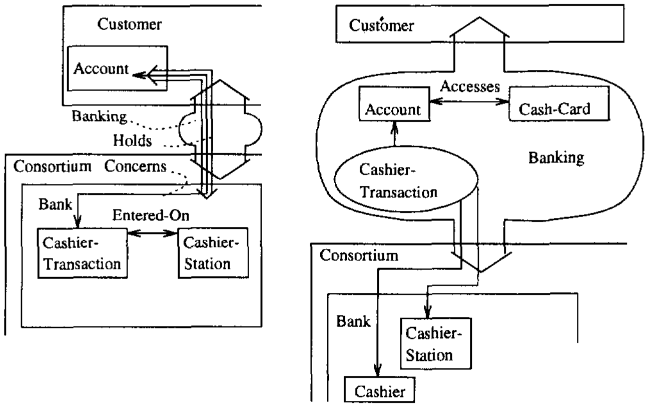
```
TheAccount.{ ... List ... }
```

The `Holds` association has a local association `Concerns` (figure 10a). This association is asymmetric because it is between the `Account` (as `Holds` is too) and the `Cashier-Transaction` local to `Bank`:

```
Banking: --class [ ... ]
( ...
  Holds: --class [ ... ]
  ( ...
    Concerns: --class
      [TheCashier-Transaction:
         Consortium.Bank.Cashier-Transaction] (...)
  ... )
... )
```

The association `Entered-On` is local to the class `Bank`. The domains of `Entered-On` are the classes `Cashier-Transaction` and `Cashier-Station`, also both local classes `Bank` (figure 10a):

```
Consortium: class
( ...
  Bank: class
  ( ...
    Entered-On: --class
      [TheCashier-Transaction: Cashier-Transaction,
       TheCashier-Station: Cashier-Station] (...)
  ... )
... )
```



a) Banking Illustration b) Illustration of Alternative Perspective

Figure 10: Illustration of Example

The Alternative Model. An enclosing association class may also have local usual classes.

In this model of the `Banking` example the classes `Account` and `Cash-Card` are still the domains of the association `Accesses` but now all these are local to the `Banking` association. `Cashier-Transaction` is a ternary association also local to `Banking` with the domains `Account`, `Cashier`, and `Cashier-Station` (figure 10b):

```
Customer: class (...)

Consortium: class
( ...
  Bank: class
  ( ...
    Cashier-Station: class
    ( ... Station-Id: method(...) ... )
    Cashier: class
    ( ... Cashier-Id: method(...) ... )
  ... )
... )

Banking: --class
[TheCustomers:* Customer, TheConsortium: Consortium]
( ...
  Account: class ( ... Balance: method(...) ... )
  Cash-Card: class (...)
  Accesses: --class
    [TheAccount: Account,
     TheCash-Card: Cash-Card] (...)

  Cashier-Transaction: --class
    [Concerns-Account: Account,
     Entered-By: Consortium.Bank.Cashier,
     Concerns: Consortium.Bank.Cashier-Station]
  ( ... Authorization-Number: method(...) ... )
... )
```

In the `Banking` class a `Cashier-Transaction` object may be instantiated by `--object Cashier-Transaction(...)` with `Account`, `Cashier`, and `Cashier-Station` objects as arguments. The set of `Cashier-Transaction`'s local to a `Banking` object, as well as various methods of the associated objects, may accessed by

```
Cashier-Transaction.
    { ... Concerns-Account.Balance
      ... Entered-By.Cashier-Id
      ... Concerns.Station-Id ... }
```

For a set of `Cashier-Transaction`'s the method `Authorization-Number` is accessed by `Cashier-Transaction.Authorization-Number`.

4 Related Mechanisms

Object–oriented modeling originates from the (simulation) models in SIMULA 67 [Dahl et al. 84]. In this modeling the inheritance mechanism, as an example, supports the specialization of concepts. Language mechanisms may be designed in general to support the abstraction processes in terms of concepts and phenomena [Kristensen & Østerbye 94].

Abstraction mechanisms may support the *logical* and *physical* view of a system. The logical mechanisms tend to be the most important ones because they express the meaning of the description. The physical mechanisms however are indispensable because they organize the description in manageable pieces and for different purposes. The two purposes have been mixed in most languages throughout the history of programming languages. It has been a problem that the distinction of these purposes has not been clarified and that the properties of the mechanisms have not been presented clearly according to this distinction. The mechanisms of this paper are clearly logical. In the following we compare our proposal with related logical as well as physical mechanisms.

Association Classes. Implicit and complex relations are not supported by the following related mechanisms:

Relations [Rumbaugh 87] and the corresponding *associations* in OMT [Rumbaugh et al. 91] are object–external abstractions and are useful for designing and partioning systems of interrelated objects. Associations may be instantiated and may have attributes, but the instances are not objects, in contrast to our work.

Contracts [Helm et al. 90] are specifications of behavioral dependencies amongst cooperating objects. Contracts are object–external abstractions and include invariants to be maintained by the cooperating objects. The focus is on inter–object dependencies to make this explicit by means of supporting language mechanisms. The result is that the actions – i.e., the reactions of an object to changes – are removed from the object and described explicitly in the contracts: The objects are turned into reactive objects, whereas the reaction–patterns for an object in its various relations with other objects are described in the corresponding contracts. The intention of the contract mechanism is not modeling of real world phenomena and their inter-dependencies. Instead the intention is to have a mathematical, centralized description, that supports provable properties. The description is mathematically rigorous. Unlike our approach the instantiations of contracts are not objects and can not have attributes, methods etc.

Enclosing Classes. The important difference between enclosing classes and the following related mechanisms is that the abstraction processes, exemplification, specialization, and aggregation are only supported by enclosing classes. The support of exemplification, interpreted as instantiation of an object from an enclosing class, implies that the enclosing object exists as an object at run–time, and as such, supports the *execution* organization. In contrast to this, most of the following mechanisms only support the *program* organization, for example by means of modules, which are only present during the development of the program and at compilation time.

Patterns (corresponding to classes) in Beta [Madsen et al. 93] may be nested for several purposes, one of which is block structure [Madsen 87]. Nested patterns are semantically similar to enclosing and local classes with respect to the existence at run–time. However, no associations (basic, implicit, or complex) are available.

The concepts *subsystems* and *contracts* [Wirfs–Brock et al. 90], which build on the concepts *responsibilities* and *collaborations*, are powerful mechanisms for understanding and expressing the relationships between classes and groups of classes. The mechanisms have no semantic influence but give additional information concerning the organization and cooperation of objects.

The *module diagram* [Booch 91] is part of the physical design of a system and describes the allocation of classes and objects in software modules as a concrete implementation of the logical design. *Subsystems* are introduced to represent clusters of logically related modules. The *class category* is an abstraction mechanism which supports the understanding of the logical architecture of a system. It has no effect on the execution of a system but supports program organization.

Nested classes (and the *friend* mechanism) of C++ [Stroustrup 91] are only related to compile–time visibility of attributes.

The *cluster* [Meyer 92] (not part of Eiffel but of Lace only) is used for arranging classes into groups. Clusters do not require specific language support, as this can be provided by the operating system facilities. Clusters support program organization.

The *subject* [Coad & Yourdon 91] is an organizational structure for programs intended to guide the reader through the description of a large complex model.

In OMT [Rumbaugh et al. 91] the *module* is a logical construct for grouping classes and associations. A *sheet* is the mechanism for breaking a large model down into a series of pages and a module consists of several sheets. Both of these mechanisms support program organization. Modules are also part of system design, which involves breaking a system into subsystems. Subsystems are neither objects nor functions, but packages of interrelated classes etc. In addition, subsystems may be organized in *layers* and *partitions.* Subsystems are part of the *architecture* of a system and all this is concerned with the physical organization of the model.

Restrictions. To simplify our description we have restricted ourselves from describing various other aspects of both enclosing and associations classes. However, the missing aspects are mostly orthogonal.

Regarding associations, we have introduced the following limitations:

- Order and multiplicity of associations: Only mechanisms for binary associations are defined and only in the form of one–to–one and one–to–many.

- Active Associations: Only passive associations are discussed. *Transverse activities* [Kristensen 93a], [Kristensen 93b] have an action part specifying a partial life cycle of the associated objects. The cooperation of objects is then described in an alternative way to object–centric method invocations.

- The abstraction processes, exemplification, specialization, and aggregation for enclosing and association classes: Specialization, interpreted as forming more *special* association classes from a more *general* association class, may include the specialization of the domains and the addition of more domains. Aggregation, interpreted as forming a *whole* association object from *part* association objects, may include the use of the attributes, methods and the domains of the part object for aggregation of the similar elements of the whole object.

Regarding enclosing objects the following aspects are not covered:

- Visibility rules for enclosing classes and the access of global attributes.
- The distinction between local object and part object.
- The movability of objects between enclosing objects and the possibility of multiple enclosing objects.

5 Experimental Project

An experiment in programming language support of enclosing classes and association classes is described in [Andersen et al. 93]. The objective was to gain more experience with the design of abstraction mechanisms of this kind, to consider efficient implementation techniques, and to be able to use the language mechanisms and the implementation for a reasonably realistic test case. The experience from the test case is that the combination of enclosing and association classes is straightforward to use and appears to give well–organized descriptions.

The design allows an object to move from one enclosing object to another. The model is based on static binding of names and the movability of objects introduces *multiple binding* of names. The concrete language mechanisms were constructed as additions to the Beta language: Environment classes may be listed in an optional clause for a class. Associations are supported by predefined classes and methods. The experience from the implementation is that movable components introduce a complex lookup mechanism for method activations [6] and that the predefined classes for associations require comprehensive underlying structures to implement the advanced functionality of their methods.

6 Summary

The underlying thesis advocated in this paper is that in existing object–oriented methodologies and description mechanisms classes and objects appear as isolated elements with very simple associations between them. However, there are other kinds of phenomena such as implicit associations between components and information existing between such components. This is important for the modeling of organization and cooperation of classes and objects. We have proposed language mechanisms in the form of nested associations and classes to support such descriptions. The main results may be summarized as follows:

- Complex associations support the modeling of the organization and cooperation of objects in object–oriented analysis, design and implementation.
- The local relation is a powerful, implicit association of an enclosing object with its local objects (and also for an association with its local associations).
- A complex association can have local associations for which the domains are local to the domains of the enclosing association.
- Associations are classes and the instances of associations are objects with attributes and methods.
- Complex associations support well–organized descriptions of the static structure as well as simple, efficient dynamic access of this structure.

---

[6] In static languages the binding of a method is usually done at run–time, dependent on which object is currently denoted. In dynamic languages methods may be added and deleted at run–time, and there is a need for dynamic lookup of the method. The lookup required in the case of multiple binding varies between different, but fixed superclass hierarchies because the object itself – or some of its enclosing objects – may have moved.

Acknowledgments. We thank Erik F. Andersen, Peter Gilling, Poul Holdt-Simonsen, Lars Milland for inspiring discussions and new insight obtained in relation to the experimental project. We also thank David Garlan for help with revising the paper.

References

[Andersen et al. 93] E.F.Andersen, P.Gilling, P.Holdt-Simonsen, L.Milland: Coherent and Well-Organized Object-Oriented Programming. Thesis (in Danish), Aalborg University, 1993.

[Booch 91] G.Booch: Object Oriented Design with Applications. Benjamin/Cummings 1991.

[Coad & Yourdon 91] P.Coad, E.Yourdon: Object-Oriented Analysis. 2/E, Prentice-Hall, 1991.

[Dahl et al. 84] O.J.Dahl, B.Myhrhaug, K.Nygaard: SIMULA 67 Common Base Language. Norwegian Computing Center, edition February 1984.

[Helm et al. 90] R.Helm, I.M.Holland, D.Gangopadhyay: Contracts: Specifying Behavioral Compositions in Object-oriented Systems. Proceedings of the European Conference on Object-Oriented Programming / Object-Oriented Systems, Languages and Applications Conference, 1990.

[Kristensen 93a] B.B.Kristensen: Transverse Classes & Objects in Object-Oriented Analysis, Design, and Implementation. Journal of Object-Oriented Programming, 1993.

[Kristensen 93b] B.B.Kristensen: Transverse Activities: Abstractions in Object-Oriented Programming. Proceedings of International Symposium on Object Technologies for Advanced Software (ISOTAS'93), 1993.

[Kristensen & Østerbye 94] B.B.Kristensen, K.Østerbye: Conceptual Modeling and Programming Languages. To appear in Sigplan Notices, 1994.

[Madsen 87] O.L.Madsen: Block Structure and Object Oriented Languages. In: B.D.Shriver, P.Wegner: Research Directions in Object Oriented Programming. MIT Press, 1987.

[Madsen et al. 93] O.L.Madsen, B.Møller-Pedersen, K.Nygaard: Object Oriented Programming in the Beta Programming Language. Addison Wesley 1993.

[Meyer 92] B.Meyer: Eiffel, The Language. Prentice Hall, 1992.

[Rumbaugh 87] J.Rumbaugh: Relations as Semantic Constructs in an Object-Oriented Language. Proceedings of the Object-Oriented Systems, Languages and Applications Conference, 1987.

[Rumbaugh et al. 91] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorensen: Object-Oriented Modeling and Design. Prentice-Hall 1991.

[Stroustrup 91] B.Stroustrup: The C++ Programming Language. 2/E, Addison-Wesley 1991.

[Wirfs-Brock et al. 90] R.Wirfs-Brock, B.Wilkerson, L.Wiener: Designing Object-Oriented Software. Prentice Hall, 1990.

Language Mechanism Summary

Basic Associations (figure 8a)

Summary: *Associations are classes which describe relations between classes, the domain classes. Associations may be instantiated as objects with attributes and methods. The objects may be denoted and accessed by special operations*

```
R: --class [roleA: A, roleB:* B] ( ... r:(...) ... )
A: class ( ... a:(...) ... )
B: class ( ... b:(...) ... )
anA: object A
aB: object B
```

Mechanism / Notation	Meaning & Remark
	R–Object Instantiation:
`--object R(anA,aB)`	An object of `R` for `anA` and `aB`
	R–Object Denotation:
`R(anA,aB)`	The object for `anA` and `aB`
`R(-,-)`	The objects of `R`
`R(-,aB)`	The `R` object associated with `aB`
`R(anA,-)`	The `R` objects associated with `anA`
`R(-,This)`	`R(-,aB)` from inside `aB`
`R(This,-)`	`R(anA,-)` from inside `anA`
	A–Object Denotation:
`R(-,aB).roleA`	The `anA` object associated with `aB`
`R(-,-).roleA`	The `A` objects of `R`
`R(-,This).roleA`	`R(-,aB).roleA` from inside `aB`
`roleA`	The `A` object from inside `R`
	B–Object Denotation:
`R(anA,-).roleB`	The `aB` objects associated with `anA`
`R(-,-).roleB`	The `B` objects of `R`
`R(This,-).roleB`	`R(anA,-).roleB` from inside `anA`
`roleB`	The `B` object from inside `R`

If no role names are specified it is possible to use the class names `A` and `B` as default role names.

A denotation of an attribute (`a`, `b`, and `r` of `A`, `B`, and `R`, respectively) has the form of an object denotation (single object or a set of objects), a dot, and the attribute name. An example of such a denotation is `R(anA, aB).r`.

Visibility restrictions may apply so that not all kinds of attributes may be accessible. Furthermore, the attributes of an object may not be visible because the object can be denoted from outside (for example `A` by `R(-,-).roleA`).

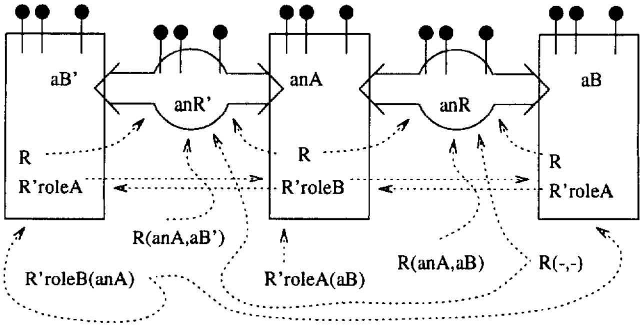


Figure 11: Illustration of Object Denotation

Shorthand Notation	Meaning
`R'is (anA,aB)`	`R(anA,aB)<>None`
`R'all`	`R(-,-)`
`R'roleB(anA)`	`R(anA,-).roleB`
`R'roleB`	`R'roleB(This)` from inside `A`
`roleB(anA)`	`R'roleB(anA)`
`roleB`	`roleB(This)` from inside `A`
`R`	`R(This,-)` from inside `A`
`R'r`	`R(This,-).r` from inside `A`
`R'roleA(aB)`	`R(-,aB).roleA`
`R'roleA`	`R'roleA(This)` from inside `B`
`roleA(aB)`	`R'roleA(aB)`
`roleA`	`roleA(This)` from inside `B`
`R`	`R(-,This)` from inside `B`
`R'r`	`R(-,This).r` from inside `B`

Local Classes and Objects (figure 8b)

Summary: *In an enclosing class a local class is a description of a collection of objects, in which the existence of such local objects are dependent on the existence of an enclosing object. The enclosing object may have attributes, methods, local objects, and associations to other objects. The local objects may be associated with objects external to the enclosing object. The local objects may also be associated by means of local associations*

```
C: class
( ...  c:(...)  ...
  Z: class ( ... z:(...) ... )
  aZ: reference Z
...)
aC: object C
```

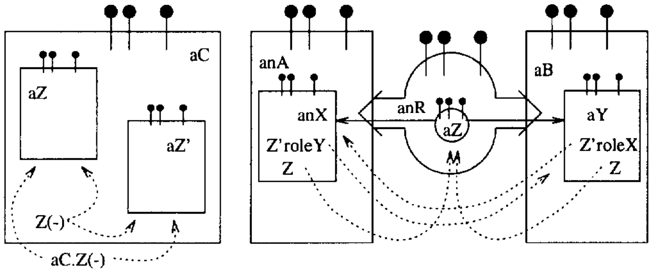
The `aC` is an enclosing object, with local `Z` objects, but it is also an object, so that `aC.c`, `aC.Z`, and `aC.aZ` may be accessed. We use the dot notation (for example `aC.z`) to simplify the description and we leave it open whether or not `z` objects should be accessible from inside `c` objects only.

`aZ` is an example of an explicit reference to a `Z` object. The class `C` (together with class `Z`) also defines an implicit association between any `C` object `aC` and the `Z` objects local to `aC`.

An optional role name may be included in the declaration of the `Z` class, for example `roleZ` in `(roleZ:Z): class(...)`.

From inside a `C` object the following mechanisms are available (may also be available from outside `C` by using the dot notation for `C` objects):

Mechanism / Notation	Meaning & Remark
	Z–Object Instantiation:
`--object Z()`	of an object local to a `C` object
	Z–Object Denotation:
`Z(-)`	of the objects local to a `C` object
`roleZ`	of the objects local to a `C` object
	z–Attribute Denotation:
`Z'z`	of the `Z` objects of a `C` object



a) Implicit Association b) Complex Association

Figure 12: Illustration of Object Denotation

Complex Associations (figure 8c)

Summary: *An (enclosing) association class is a relation between (enclosing) domain classes. Enclosing association classes have local classes and local association classes. The existence of objects of the local associations are dependent on the existence of an object of the enclosing association. The domains of the local associations are either local classes to the enclosing association class or local classes of the domain classes of the enclosing association*

```
R: --class [roleA: A, roleB: B]
(  ...  r:(...)  ...
  Z: --class [roleX: A.X, roleY: B.Y]
  ( ... z:(...) ... )
... )
A: class ( ... X: class(...) ... anX: object X ... )
B: class ( ... Y: class(...) ... aY: object Y ... )
```

`R` is an association class, with a local association class `Z`. `A` and `B` are classes, with local classes `X` and `Y`, respectively. `Z` is a relation between `X` and `Y`.

The class `Z` is also an implicit association for an `R` object and the `Z` objects local to this.

From inside `R`, `Z`, `X`, and `Y` the following mechanisms are available (may also be available from outside these classes by using the dot notation for `R`, `Z`, `X`, or `Y` objects):

Mechanism / Notation	Meaning & Remark
	Z–Object Instantiation:
`--object Z(roleA.anX,roleB.aY)`	from inside `R` (or `X` or `Y`)
	Z–Object Denotation:
`Z(...)`	from inside `X` of `A` (`Y` of `B`)
	X–Object Denotation:
`Z(...).roleX`	from inside `Z` of `R`
`roleX`	from inside `Y` of `B`
	Y–Object Denotation:
`Z(...).roleY`	from inside `Z` of `R`
`roleY`	from inside `X` of `A`
	z–Attribute Denotation:
`Z'z`	inside `X` of `A` (`Y` of `B`)

The notation `Z(...)` stands for all the possibilities available for accessing association objects, similar to the possibilities for basic associations.

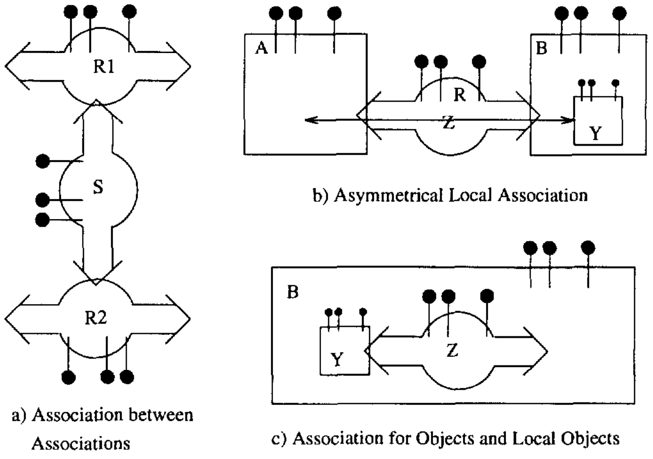
Association between Associations (figure 13a)

Summary: *Associations may exist between associations, so that the domains may be associations at any level*

```
R1: --class [A1, B1] (...)
R2: --class [A2, B2] (...)
S: --class [roleR1: R1, roleR2: R2] (...)
```

In the instantiation of an association between two existing objects of `R1` and `R2` we may use references to association objects, by the notation `--reference R`:

```
anR: --reference R
anR := --object R (anA, aB)
```



a) Association between Associations

b) Asymmetrical Local Association

c) Association for Objects and Local Objects

Figure 13: Illustration of Associations

Asymmetrical Local Association (figure 13b)

Summary: *Local associations may exist between the domain of an association and local objects in the other domain(s)*

```
R: --class [roleA: A, roleB: B]
   ( ... Z: --class [roleY: B.Y] (...) ... )
A: class (...)
B: class ( ... Y: class(...) ... aY: object Y ... )
```

The instantiation of an association between `anA` and `aB.aY` is denoted `--object Z (aB.aY)` from inside `R`. From inside `A` the object denotation of `aB.aY` has the form `Z(this).roleY`.

Association for Local Objects (figure 13c)

Summary: *Explicit associations may exist between an object and some local objects of this enclosing object*

```
B: class
( ... Y: class (...) ... aY: object Y ...
  ... Z: --class [roleY:* Y] (...)
...)
```

The instantiation of an association between `aB` and `aB.aY` is denoted `--object Z(aY)` from inside `B`. The denotation of all `aY` objects in the association `Z` from inside `B` has the form `Z(-).roleY`.