Object Acquaintance Selection and Binding

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www: http://www.pt.hk-r.se/~bosch

Abstract
Large object-oriented systems have, at least, four characteristics that complicate object communication, i.e the system is distributed and contains large numbers, e.g. thousands, of objects, objects need to be reallocated at run-time and objects can be replaced by other objects in order to adapt to the dynamic changes in the system. Traditional object communication is based on sending a message to a receiver object known to the sender of the message. At linking or instantiation time, an object establishes its acquaintances through name/class based binding and uses these objects through its life time. If this is too rigid, the software engineer has to implement the binding of objects manually using pointers. In our experiments we found the traditional acquaintance communication semantics too limited and we identified several problems, related to the reusability of objects and selection mechanisms, understandability and expressiveness. We recognised that it is important to separate an class or object's requirements on its acquaintances from the way an object selects and binds its acquaintances in actual systems. Based on this, we studied the necessary expressiveness for acquaintance handling and identified four relevant aspects: type and duration of binding, conditions for binding, number of selected objects and selection region for binding. To implement these aspects, we defined *acquaintance layers* as part of the *layered object mode*l. Acquaintance layers uniformly extend the traditional object-oriented acquaintance handling semantics and allow for the first-class representation of acquaintance selection and binding, thereby increasing traceability and reusability.

1 Introduction

Object-oriented systems are constantly growing in size and complexity. A large system might easily contain thousands of objects on a distributed hardware platform, often incorporating heterogeneous hardware. The objects in the system should be able to move from one site to another when the need arises, e.g. due to load balancing or site failure. Also, to support evolution of the system, objects need to be replaceable by other objects.

One example of such a system can be found in modern electric distribution networks [Enersearch 96]. Future electricity distribution networks will be different in many respects, but an important difference is the notion of two-way communication between electricity producer and electricity consumer. A second difference is that every electrical appliance will be extended with a microprocessor that controls the appliance. An average household will contain tens of these microprocessors and consequently a few hundred objects controlling the house and its appliances. In addition to the small microprocessors associated with electrical appliances, households will contain larger computers for more advanced, e.g. coordinating, tasks. Although most objects in the system need to be kept at particular processors, several objects can and need to dynamically change location depending on external events and circumstances. For instance a user object will follow with a physical person and automatically change the conditions in the room that is entered or left. Also, new electrical appliances, when put into the electricity network, need to automatically bind themselves to the acquaintances they need. Finally, the electricity producer is responsible for certain objects in the system, e.g. the objects for buying electricity. Sometimes, these objects will need to be replaced with updated versions. As it is virtually impossible to bring the system down and reboot with a new version of the software, these changes have to be incorporated in run time. The aforementioned aspects require that objects can select and bind other objects in a flexible and dynamic fashion.

Object-oriented languages use name-based binding for connecting objects. Name-based binding can be done at compile- or link-time, or during run-time as in Smalltalk. Approaches such as Corba [OMG 96] and OLE/COM [Microsoft 96] add functionality for run-time name-based binding and distribution to languages such as C++. Distribution proved to be a problem for the traditional object binding approaches since memory-based pointers could no longer be used. As a solution, the software engineer was forced to distinguish between objects in the same memory space and

objects in other memory spaces. The uniformity of this approach is clearly less than optimal and the software engineer, in some way, has to implement the interaction with remote objects manually.

Regardless of distribution, name-based binding is too rigid in many situations. For this reason, most object-oriented languages provide some form of object pointers. Object pointers provide a primitive mechanism that allows the software engineer to hand-code customised object selection and binding mechanisms. The problem of pointers, however, is that the specifications for selecting and binding an acquaintance object becomes an integral part of the class specification and that the object nor the selection and binding mechanism can be reused separately.

The name-based object binding and the manual object selection which is programmed as part of the class specification increase the dependence of the class on the environment in which it is used. If the class should be reused in a context different from the context for which it was originally defined the name of one of its acquaintance objects might be different as well as that the binding procedures for some of its object pointers might be different. If the selection and binding is embedded in the class code, this reduces reusability considerably.

In this paper, we study the problem of acquaintance selection and binding for objects in the new class of objects systems that is appearing: large, distributed and open object systems in which objects are constantly created, moved around and removed from the system. The traditional notion of an application as a closed, specialised entity with its own data stored in its own database and a tailored user-interface is rapidly being replaced with another perspective in which objects operate in large, distributed systems and make use of offered services and work in an integrated manner with other objects. Traditional applications only have vertical interfaces, i.e. interaction with the operating system and the user-interface system, whereas the new type of applications, in addition to the vertical interface, also has a horizontal interface, i.e. an interface to other applications and parts of other applications. The conventional object model provides only limited features to deal with the requirements imposed by these new application types and its use may lead to a number of problems with respect to acquaintance handling. As a solution to these problems, we present an extension in the form of a new layer type `Acquaintance`, to the layered object model (**LayOM**), our object-oriented research language, that provide solutions to the identified problems.

Our approach to a solution of the acquaintance handling problems is to explicitly distinguish between the specification of the various acquaintances required by an object and the actual selection and binding process in which one or more instances play the role of an acquaintance for an object. By separating these two aspects of object acquaintances, one can specify classes as general as possible and define for individual instances of the classes how the required acquaintances should be selected and bound.

For the discussion of acquaintance handling in this paper, we have adopted a hierarchical object system view which is different from the traditional view. Since the size of the object system that we described before will be like thousands of objects, the traditional ‘sea of object’ approach is really not suitable. Even when one distinguishes between local and remote objects, the number of remote objects will be too large to handle in an efficient way. It is necessary to impose some structure on the object system, and the hierarchical structure is a very appropriate approach. This hierarchical grouping of objects is also used in object-oriented analysis and design through the notion of subsystems. A large application can recursively be decomposed into subsystems, leading to a hierarchical structure. However, at the implementation level no corresponding concept is available, leading to two possible approaches. Most software engineers do not implement subsystems and the resulting situation is the ‘sea of objects’ in which the software engineer has to specify which objects communicate with each other. The other approach which is taken is that subsystems are implemented as composed objects, encapsulating the objects in the subsystem. The advantage of this approach is that the objects are organised in a hierarchical structure, which improves the overview over the system. An important disadvantage, however, is that since the objects are fully encapsulated in the subsystem, but still need to communicate with objects in other subsystems, the subsystem objects need to implement considerable amounts of methods that only forward messages to objects contained in them. Thus, subsystems have different requirements on encapsulation than objects and both not implementing subsystems and implementing subsystems as composed objects lead to problems. As a solution, we allow objects to extend their interface, which conventionally only consists of public methods, with part objects. So, part objects that are explicitly specified to be visible from outside the encapsulating object can be called directly by other objects. By providing flexible encapsulation, it becomes much more feasible to represent subsystems at the implementation level as composed objects that can partially be seen through. However, these aspects are currently under investigation and will be described in a future paper.

The remainder of this paper is organised as follows. In the next section, our view on the object-oriented system thinking is presented that provides a context and justification for the work presented in this paper. In section 3, the problems that we identified with acquaintance selection and binding in the conventional object-oriented model are

described. Subsequently, in section 4, the expressiveness that we believe to be necessary for dealing with object acquaintances is described. In section 5 the layered object model is introduced and the extensions that deal with object acquaintances are described. Section 6 is concerned with related work and the paper is concluded in section 7.

2 System Overview

Whereas traditional applications of object-oriented systems have been closed and relatively static in their nature, a new class of systems is arising in which necessary properties are openness and dynamicity. Other characteristics of these new systems are distribution and their large size, both in the number of objects and in the geographical coverage of these systems. Parts of applications such as object communication that previously were hard-coded in the class code need to be defined in a more flexible and expressive manner.

An important aspect is the notion of communication between cooperating objects. The binding of cooperating objects traditionally was performed statically during the compilation or linking stage or dynamically through passing object references. Since selecting and binding an acquaintance is a more complex and dynamic process in large, distributed systems, additional functionality is required from the object-oriented language in which these systems are implemented. In these distributed systems, the number of possibilities to select an object to communicate with is much larger. Consequently, more expressive mechanisms for specifying acquaintance handling are required.

Before an object can communicate with an acquaintance, three aspects have to be fulfilled. The first is that the object contains a *specification* of the requirements it puts on its acquaintance, i.e. the acquaintance has to fulfil certain requirements, e.g. be an instance of a particular class. Secondly, at some point, be it at the time of definition, linking or instantiation or when the object tries to send the first message to the acquaintance, a *selection process* has to be performed to select the acquaintance from the objects in the context of the object. Finally, a *binding step* has to take place in which the object binds itself to the acquaintance and uses the acquaintance for its communication. These steps can be identified in virtually all module-based systems, but the way these steps are performed is often embedded and hard-coded in the language implementation, leaving no flexibility to the software engineer.

To understand the communication mechanisms required in the aforementioned new class of systems, the characteristics of large, distributed systems need to be described. The following characteristics of these systems we consider relevant for object communication: the notion of system, hierarchical object organisation and object group communication. To justify the relevance of these characteristics, three example system domains where acquaintance handling is an important issue are described in the next section. Subsequently, in section 2.2, the architecture of large object systems, as used in the remainder of the paper, is discussed.

2.1 Example Systems

The issues of acquaintance handling addressed in this paper are perhaps primarily relevant for open, dynamic applications, although also traditional, closed applications can benefit. In these traditional applications, the references between objects often are rather static during the execution of the program. The need for more expressive communication constructs is, consequently, not so acute. However, in current and future application domains, several authors have recognised a need for more expressive communication constructs. In the following sections, three application domains in which we have experienced the need for flexible object communication are described.

2.1.1 Integrated manufacturing systems

Manufacturing systems have already reached some level of integration. Especially at the level of the production cell, there is a rather high level of integration. The different equipment parts of the production cell communicate frequently and perform tasks together. The equipment in the production cell has hard real-time tasks, such as the control of a robot arm, soft real-time tasks, such as changing a tool in a machine, and non real-time computation, such as the generation of the production figures for the last hour.

Already today and even more so in the future, manufacturing systems are becoming more and more integrated in their architecture. This is necessary to achieve the flexibility and openness that is required by the new demands on manufacturing systems, such as efficient and cost effective production of small series and individual products. In such an *integrated* manufacturing system, the production demands on the system and its structure will be constantly changing. This flexibility will also affect the communication between the objects in the system.

The software part of a manufacturing system can be viewed as a large, distributed system in which the relevant entities in the factory are represented by objects. Several objects, e.g. objects representing the products under manufacturing, are changing context rather frequently and therefore need to select and bind themselves to new objects after every context change. The manufacturing system is supposed to be highly flexible, meaning that production cells can dynamically be added to and removed from the system. The system has to dynamically reconfigure itself and product objects need to start using the functionality provided by the new production cells automatically. Unless one prefers to explicitly program this behaviour, more expressive acquaintance handling constructs are required to deal with these types of behaviour.

2.1.2 Electricity distribution automation

Future electricity distribution networks will be different in many respects, but one major difference will be the notion of two-way communication between electricity producer and electricity consumer. A second difference is that every electrical appliance will be extended with a microprocessor that controls the appliance. An average household will contain tens of these microprocessors and consequently at least a few hundred objects controlling the house and its appliances. In addition to the small microprocessors associated with electrical appliances, households will contain larger computers for more advanced, e.g. coordinating tasks. Although some objects in the system need to be kept at particular processors, most objects can and need to dynamically change location depending on external events and circumstances. In addition, new appliances can be added dynamically and the objects in these appliances need to automatically select and bind their required acquaintance objects. An additional problem is that the electricity producer is responsible for certain objects in the system, e.g. the objects for buying electricity. Sometimes, these objects will need to be replaced with updated versions. As it is virtually impossible to bring the system down and reboot with a new version of the software, these changes have to be incorporated in run time. This requires that an object can be replaced by another object without disturbing the clients of the object.

2.1.3 Mobile telecommunications

In the domain of telecommunications, especially mobile communication is a large and increasing market. The infrastructure required for mobile telephony, however, is more complex than for traditional telephony. This is primarily due to the dynamicity in the location of the phones. For example, a physical mobile phone is represented by a mobile phone object in the distributed information system of a mobile telephony operator. A mobile phone object has several acquaintances, among others a base station object, an operator object and a voice mail object. Since the owner of the physical phone moves around, the phone object also moves through the distributed information system. In some cases, the phone user travels outside the region covered by the operator, in which case another operator that covers the particular region provides telephone services for the phone, under the condition that the two operators have such an agreement. Another example is the situation where the user of the phone is moving during a phone call. This may require that the phone changes base station, requiring the phone object to be moved from one base station object to a remote base station object. The consequence of these types of behaviour is that the mobile phone object needs to reallocate and to reselect and re-bind some of its acquaintances. For instance, the base station acquaintance needs to be rebound after every relocation and needs to be selected from the immediate context of the object. The operator object only needs to be rebound if the new base station object is owned by a different operator, whereas the voice mail object is bound permanently to the object.

2.2 System Organisation

Based on the three example domains mentioned in the previous section, but also other examples not mentioned in this paper, one can abstract a number of properties of open, dynamic object-oriented systems. Important issues while discussing such systems are the system concept itself, the organisation of objects and other entities and the communication between objects.

2.2.1 System Concept

The notion of a system with closed boundaries and single instantiation time is no longer valid in the systems addressed in this paper. The ‘system’ is becoming more of a context or a space in which objects are created, changed and removed constantly. Since these systems are so flexible and dynamic, the different entities in the system cannot, up to the same degree, create permanent dependencies on each other. Since these systems generally are distributed, they are

also dynamic at the level of physical entities. For example, nodes in the system network can be added and removed dynamically. An illustrative, though extreme, example of the type of system we are referring to is *Internet*. Internet cannot be considered as a system in the traditional meaning of the word. No single person has complete overview over the system or has the right to reorganise the system. Similarly, in *intranets*, i.e. intra-organisational networks, generally no single person has complete control over and understanding of the system

A consequence of the different system view can be found in the way applications are instantiated and configured in the system. In the traditional systems, applications have only a 'vertical' interface, in that these applications only communicate with the underlying operating system and with the user interface on top of the application. The interface of the application to the entities it interact with is very rigid, since both the operating system and the user interface have well-defined, static interfaces on which one can rely without limiting the useability of the application. The few dynamic aspects of the application, e.g. as a consequence of different versions of the operating system or user interface, could be hand-coded into the application without too much overhead.

The development in the open flexible object systems is that the application has, next to the vertical interface, also a horizontal interface that tends to be more important than its vertical counterpart. The horizontal interface consists of references to and communication with objects external to the application. These objects can be data or service providers, but they can, in principle, represent any type of behaviour.

The notion of an application, in these systems, is no longer an independent entity executing separated from everything else in the system. Whereas an application previously formed an impenetrable encapsulation boundary for the entities that it contained, its role in the new systems is limited to two aspects, i.e. to represent a unit of instantiation and, perhaps, removal and, secondly, an encapsulation boundary for security means. The first role is to instantiate a group of objects that combined form a relevant entity as a single action. These objects, however, will in general communicate with each other, but also with other objects outside the application. That leads to the second role of the application, i.e. to provide an encapsulation boundary. The encapsulation boundary can be used to restrict access to certain objects contained in the application, or to restrict access by certain clients. However, communication between external objects and internal objects is necessary requirement, so the encapsulation cannot restrict all access by external objects.

2.2.2 Hierarchical object organisation

If one takes the system perspective in the previous section, it becomes clear that the objects in that system need to be organised in a structured manner in order to be able to deal with the large number of objects. The traditional 'sea of objects' approach where all objects in the system are visible to each other and exist at the same level is infeasible. Although this seems an obvious requirement from a design perspective, fact is that, although software engineers make use of subsystems, very often the implementation is performed in a sea of objects approach. On the other hand, for inter-application communication, the traditional approach is to take a hand-coded interfacing approach, causing the application to treat objects external to it very different from objects contained inside the application.

The lack of uniformity in treating internal and external objects is obviously problematic for several reasons. One reason is that an object that previously used an internal object and needs to change to using an external object requires considerable adaptation of the code. A second reason is that the interface generally only provides access to those features required at application development time. The intra-application object communication, as described, is often based on a 'sea of objects' approach. The disadvantage of this approach is, at least, twofold, i.e. complexity and access control. An average application can easily contain hundreds of objects, leading to problems of complexity when searching, replacing and accessing objects. Secondly, since each object is directly accessible by all other objects, the object might be called by clients that should not have access to that object. A solution would, of course, be to program access control checks into each object, but that leads to considerable implementation and execution overhead and reduces the reusability of the resulting object dramatically.

To address the aforementioned problems, a unified object organisation is required that deals with the problems of communication with external as well as internal objects. The approach that is used in this paper is to organise objects in an hierarchical manner. Each object is located inside another object, that, in turn, together with other objects is encapsulated by a higher-level object. This process can continue for some levels before a top level is reached at which the complete system is represented as a top-level object. Although this organisation may seem very hierarchical, most systems fit the description. For example, an distributed object-oriented system contains at least two sites at which objects can execute. In virtually all situations, the objects in the system will distinguish between objects located at the same site and objects on a different site. This consequently leads to a two level hierarchical organisation. However, it is important to note that the hierarchical object organisation should be based on the logical structure of the system and

not on its physical organisation. Higher level composed objects in general represent entities like subsystems and not the set of objects located at a particular workstation. Nevertheless, there is, in certain cases, a relation between an object and a physical entity, but this issue is not discussed here.

From the aforementioned perspective, the complete system can be viewed as a single, large object. This object consists of nested objects, that recursively may be composed of nested objects. This system organisation provides more structure, when compared to the traditional object-oriented model in which nesting is not explicitly offered as a system organisation property. In these systems, all relevant objects generally are in the same name space, leading to a potentially very large number of objects that can 'see' each other.

However, modelling the parts of the system as normal objects leads to problems with accessing objects that are in different subsystems. If the subsystem is represented as a traditional object, all objects contained in the subsystem are encapsulated and hidden for objects in other subsystems. Although this property is very useful in object-oriented programming, it is not as appropriate in large objects for representing subsystems. If the functionality of objects in the subsystem needs to be accessible by objects in different subsystems, the subsystem, in the traditional object model, would need to put methods on its interface that forward requests by external objects to the appropriate object in the subsystem. In case of large or deeply nested subsystems, this behaviour is rather tedious since potentially many methods have to be implemented and, in case of nested subsystems, the same method may have to be defined several times, once for every encapsulating subsystem.

A solution to this problem would be some notion of *flexible encapsulation*, but we are investigating this issue. The concept builds on the ability of objects to make nested objects visible to external clients. Based on this, a subsystem can make a subset of the encapsulated objects visible and encapsulate all other objects. The flexible encapsulation concept solves the aforementioned problems related to encapsulation by subsystems.

2.2.3 Object group communication

Object communication in general considers a sender of a message and a known receiver of the message. However, in several situations a message involves operations at several objects. Examples of this type of behaviour are multi-casts and update messages in model-view-controller structures. In both cases, a message, e.g. notifying a state change, will be send to multiple objects, which are selected based on some criterion.

Object group communication requires some form of object set that is used for selection of objects for, e.g. multi-casts. The hierarchical object structure, discussed in the previous section, provides object sets, since each object is encapsulated in some composition object representing, e.g. a subsystem, and the object encapsulated in the composition object can be used as the selection set. An additional aspect of the composition objects is that the composition object itself is encapsulated in yet another encapsulation object. This leads to nested contexts and an object that intends to perform a multi-cast or another form of object group communication can decide whether only the immediate context of the object is used, the complete system or some level in between, i.e. the ***n*** encapsulating contexts. Such selections can be based on any property, including the object type, and are rather similar to queries in object-oriented databases.

Although object group communication is not supported in the traditional object-oriented model, it has been studied by several authors. Most authors are concerned with representing complex communication, or interaction, patterns between a group of objects where each object plays some roles in the pattern, e.g. [Helm et al. 90] [Aksit et al. 93] and [Pintado 95]. All these approaches are concerned with representing an interaction pattern between a group of objects as a separate entity that interacts with the involved objects. This is different from what we are concerned with in this paper; it is our aim to represent the acquaintance selection and binding process of an individual object in an expressive and reusable manner, even though this communication may be towards a group of objects.

3 Problems in Object Communication

The traditional object-oriented model only supports communication between a sender object and a known receiver object, i.e. the sender has to know the identity of the receiver object before sending a message to it. As described in the introduction, additional mechanisms for communication are required in large systems containing many objects. The communication mechanisms required for object communication can be categorised into two categories, i.e. communication with single objects and group based object communication.

In general, an object communicates with a number of other objects while performing its behaviour. These objects are necessary for the correct operation of the object and we refer to these objects as the *acquaintances* of the object. For an object to communicate with an acquaintance, three acquaintance handling aspects have to be specified. The first aspect are the *requirements* that the object puts on an acquaintance. A requirement often found in programming languages is a combined requirement where the acquaintance has to have a particular name and be of a particular class. The second aspect is a specification of how the selection of the appropriate acquaintance candidate(s) from the objects in the context of the object should be performed. For example, candidates can be searched in the immediate context of the object or globally in the system. The third aspect is a specification of how the selected candidate is bound as an acquaintance of the object. The acquaintance may be bound permanently, but the binding may also be temporary, e.g. just for a single call. Thus, we defined acquaintance handling as consisting of a requirement specification, a selection process specification and a specification of the binding type.

In traditional object-oriented languages like C++, an object is connected to its acquaintances, in one of two ways: *automatically* at compile, link or instantiation time or *manually* during run-time through explicit initialisation of references to other objects. Generally, a conventional object-oriented language provides only one way of automatically dealing with acquaintance specification, selection and binding. If this is not sufficient for the situation at hand, the software engineer is forced to manually program acquaintance handling. In table 1, the *automatic* acquaintance handling mechanisms available in the most popular object-oriented languages, i.e. C++, Smalltalk-80 and Java, are described.

Language	**specification**	**selection**	**binding**
C++	name and class	‘exactly one’ model; error in case of conflicts	link time
Smalltalk-80	name in case of global object or instance variable; reference passing otherwise	‘exactly one’ model; no notion of hierarchical contexts	definition time (warning); otherwise at first invocation (error)
Java	name and interface type	only selection within class	compile time

Table 1. Automatic acquaintance specification, selection and binding in C++, Smalltalk-80 and Java

All languages provide, either implicitly or explicitly, a pointer mechanism in addition to the name or name/class-based binding, i.e. the manual approach. However, we view pointers as a language mechanism that is provided for situations where the language mechanisms for acquaintance handling lack expressiveness, since the software engineer is forced to explicitly program acquaintance handling. That means that pointers are used whenever the language does not allow the software engineer to the express the required acquaintance handling in terms of the language. Although we agree that it is impossible to deal with all situations and that some form of explicit reference passing is required, we are convinced that more expressive automatic acquaintance handling mechanisms should be investigated. In section 4, we discuss the requirements on acquaintance handling that we consider relevant.

We have observed acquaintance communication in applications that involved both single object and object group communication. Since the traditional object-oriented languages do not provide more expressive language support for acquaintance handling, we have identified a number of problems. In the following subsections we discuss the most important problems. These problems are related to the reusability of objects and acquaintance selection mechanisms, understandability and expressiveness.

3.1 Object reusability

One important part of a class definition is the specification of the acquaintances, required by an instance of the class for its own correct operation. However, the selection and binding process may differ for the different instances of the class. If the software engineer also has to specify explicitly in the class code how to select and bind acquaintance objects, the reusability of the class is decreased because it cannot be reused in situations where the selection and binding of acquaintance objects is different, even though the functionality of the class might be applicable to the application. However, traditional object-oriented languages do not allow the software engineer to separate the requirement specification from the selection and binding process specification, leading to decreased reusability of the class code.

For example, in the C++ code below, a subclass `Product` of class `ProductionItem` is defined. Class `Product` requires an external object that contains the production schedule for the object. In this application, the name of the external object is hard-coded as `myManager`. If the class would be reused in different context, one requirement on the context of

the instances of the class would be that an object `myManager` of class `ProductionManager` exists and can be bound to at link time. This obviously decreases the reusability of the class. One would like to configure each instance of the `Product` class with binding to an object that plays the role of the production manager.

```
extern ProductionManager myManager;

class Product : ProductionItem {
   int nextProductionStep() {
      ProductionStep prodStep;

      prodStep := myManager.nextProductionStep();
      prodStep.execute(this);
   }
   ...
}
```

Alternatively, `Product` objects could store a reference to its production manager object. At instantiation or later, a reference would need to be passed to the object that points to the production manager. As mentioned earlier, the use of pointers increases the flexibility of object bindings. However, this requires the software engineer to program the object bindings manually, which requires additional programming effort, causes increased code size and increased complexity. In the situation where the product may change production manager during its lifetime since it is moved between different production sites, a potentially complex reference updating scheme needs to be programmed in the product and production manager classes. This may, in turn, reduce reusability, since, in a new context, the product class may require a different updating scheme, but since the updating is embedded in the class definition, replacing the scheme may require more effort than redesigning the class from the scratch. In addition, due to the flexibility of the binding, the object may need to test for all uses of the reference whether it is currently valid, which decreases understandability, as can be seen from the code below. The two statements from the method shown above have to be embedded with an if-statement structure to make sure that the reference to the production manager is valid.

```
   if (myManager) {
      prodStep := myManager.nextProductionStep();
      prodStep.execute(this);
   } else {
      return FAIL;
   }
```

This problem could be addressed by using a placeholder object for `myManager` in all situations where the reference does not have a valid value. This placeholder object has to be of the same type as the actual acquaintance, but the implementation should be very different. The placeholder object has to bind the reference to a suitable acquaintance object or just return without performing the intended computation. Another solution that could be used is to put the acquaintance selection in a separate method. The call structure shown above would then look as follows:

```
   this->myManager()->nextProductionStep();
```

Although both solutions separate the acquaintance handling from the method code, some problems remain. The acquaintance handling method still requires manual coding of the acquaintance handling for each acquaintance of the object, causing unwanted implementation overhead. In addition, when the object is reused through composition rather than through inheritance, the acquaintance handling method cannot be replace separately. With respect to the placeholder object, one has to note that the amount of implementation code may be considerable since the placeholder has to support the same interface as the acquaintance. Concluding, although some partial solutions could be constructed within traditional object-oriented languages, reusability of the object or the acquaintance handling remains problematic. In addition, the understandability of the code is typically not improved by these approaches.

3.2 Selection mechanism reusability

In certain situations, an object needs to select an acquaintance from a collection of objects. The acquaintance often needs to fulfil some particular requirements and there might be no, one or more objects that can provide the required functionality. For example, a product instance may need a particular type of machine to deliver a type of manufacturing step on the product, e.g. milling or welding. Since the product, while it is in production, is located in the context of a production site, several production machines may provide the required operation, but the quality and cost associated for each machine may be different. In such situations, a property-based selection process is required to select the most appropriate candidate acquaintance. This, however, requires some selection algorithm or mechanism. If this algorithm

is part of the code of the class, the algorithm is inherently not reusable. In order to be reusable, the selection algorithm needs to be specified separate from the object that uses it.

For example, in the case of the product class, its instances may need to select production machines depending on the system in which the product class is reused and on the particularities of the instances. For different instances, the balance between the factors time, cost and quality will be different, leading to different selection algorithms. If the selection algorithm, as shown in the code, is mixed with the code of the class, it is difficult, if not impossible, to reuse the algorithm separately.

```
class Product {
   ...
   executeStep(ProductionStep step)
   {
      ProductionMachine *machine;

      ... selection algorithm (machine gets a value)...
      machine->executeStep(this, step);
   }
}
```

This problem has been identified by others also. For example, the *Strategy* design pattern [Gamma et al. 94] intends to address, among others, this problem by modelling the algorithm as a separate class that is used as a part object of the object using the algorithm. The algorithm can then be replaced by replacing the part object. Our experience is that, although the design pattern works well in the general case, it is less useful for acquaintance selection. This is because the way an object gets access to the set of potential acquaintances, i.e. objects from its context, is only partially generic. Each situation also requires some very application-dependent specifications and the resulting strategy needs to be a mixture of the generic and the application-specific behaviour. For instance, the way an acquaintance is selected from a set of objects may be generic, but the condition for selecting the appropriate acquaintance is very application specific. When using a placeholder object or a separate acquaintance handling method, the generic and application specific aspects need to be mixed leading to poor reusability of the resulting object or method.

3.3 Understandability

Aspects such as understandability and intuitiveness are rather vague and often subjective terms, but one can state more concrete matters. In general, the software engineer has a particular idea of the behaviour of a class or method. If a method, in addition to the code for its basic functionality, contains code for selection of the appropriate acquaintance(s) to communicate with, the actual functionality is obscured and understandability is decreased. This is worsened by the fact that the code for the functionality and the code for acquaintance selection often are mixed together. For example, before the object can send a message to an acquaintance, it first has to check whether the pointer has an appropriate value and, if not, the binding procedure has to be invoked to provide a value. Only then a message can be sent. In addition to obscuring the functionality of the method, the way an acquaintance is selected and bound is also obscured and less clear for the software engineer.

Using separate acquaintance handling methods provides a solution to the mixing of method behaviour and acquaintance handling, but since the method contains an imperatively programmed algorithm, it may still be difficult to understand how the object selects and binds it acquaintances. The placeholder approach provides an even better separation of application domain behaviour and acquaintance handling since both are separated into multiple objects. However, in the situation where the handling of multiple acquaintances need to be represented as placeholder objects and, perhaps, also other aspects of the object are represented as separate strategy objects, the resulting structure will be very complex and difficult to understand for the software engineer. Since reuse generally begins with an evaluation of the class by the software engineer to see if it fits the intended purpose and that the code often is inspected, one can deduce that understandability forms a crucial factor for reuse.

3.4 Lack of expressiveness

As mentioned earlier, traditional object-oriented languages support only a single mechanism for binding objects, e.g. name-based binding. If the software engineer requires more flexible selection and binding mechanisms, the only alternative is to explicitly program the binding using pointers and pointer passing. Since the use of acquaintances is spread out over the class, the acquaintance binding and intended object behaviour become intertwined to the extent that they

cannot be separated. As an alternative, one could use placeholder objects or separate acquaintance handling methods, but in the previous sections it was identified that these also suffer from problems of reusability and understandability. The fact that the software engineer has to explicitly acquaintance handling indicates that the expressiveness of the associated language mechanisms is too restricted. Especially in large, distributed and flexible systems, the mechanisms for object communication available in the conventional object-oriented paradigm are too restricted in their expressiveness.

As a solution, more expressive language constructs for acquaintance handling are required so that the software engineer only has to use the pointer mechanism in a minority of the situations, instead of, as it is currently, in most situations. The question that then has to be answered is what expressiveness the acquaintance handling constructs should have. In section 4, several aspects of acquaintance selection and binding are studied.

4 Aspects of Acquaintance Handling

The communication between objects is, in the traditional object-oriented paradigm, based on direct communication between a sender and a receiver object, where the sender knows the identity of the receiver. As we discussed in section 3, this type of object communication is insufficient for communication in large, open systems. Since we aim at providing a solution to this problem, it is important to understand what expressiveness is required from language constructs dealing with acquaintance handling. In this section, four aspects of acquaintance handling are described that play a role in the communication an object and one of its acquaintances. These aspects are the time and duration of binding an acquaintance (or acquaintances), the condition based on which a particular acquaintance is selected, the number of objects acting as an acquaintance and the size of the region in which the acquaintance is searched. It should be noted, however, that, although we believe that these aspects are important and cover a large part of the required acquaintance handling expressiveness, it is possible to identify other aspects that could play a role and should be incorporated also. One should note however that it is not our intention to remove the notion of references from the programming language, but rather to drastically reduce the number of situations in which the software engineer has to manually program the acquaintance handling using these references. Secondly, all requirements on the expressiveness of programming languages are changing over time. Due to this, we believe that programming languages should be extensible. Thus, if new required expressiveness for acquaintance handling is identified, one could incorporate it in the language model. In [Bosch 95c] we refer to this principle as *paradigm extensibility* and the layered object model, presented in section 5, in which we present our solution to acquaintance handling is an *extensible* language model. In the following sections, the relevant aspects of acquaintance selection and binding are described in more detail.

4.1 Type and Duration

Virtually any object has one or more *acquaintances*, i.e. objects it communicates with in the course of its operation. The methods of the object call acquaintances through message sends of the form `acq.method(args);`. However, the method is generally specified in a class and the designer of the class cannot decide on the object to be used for communication. In traditional object-oriented systems, the selection of the actual object to communicate with is either based on compile-time or run-time binding based on the object name or by references passed to the object. As discussed in section 3, this leads to several problems because the traditional binding approach is too rigid for objects in complex, dynamic systems. The use of references solves the problem of rigidity, but has as a disadvantage that the software engineer has to manually program the object communication using the reference mechanism which is tedious, error-prone and lacks reusability.

When studying the object binding in complex, distributed systems, we have found three types of object binding. Below, we describe each type in more detail.

- **Permanent binding**: Permanent binding is, as the name implies, the situation where an acquaintance of an object is bound either at instantiation time of the object or when the first call to the acquaintance is issued. The binding to that acquaintance object is kept throughout the lifetime of the object. This type of binding seems similar to the traditional binding process, but it actually is a generalisation. In section 4.2, the conditions for selecting acquaintances are described. These conditions can include an object name but also various other features of an object.

- **Per call binding:** Since the objects in the context of an object may change dynamically, at different times, different objects may be the most appropriate acquaintance to deliver a service to the object. As an example, while travelling by car one needs a gas station from time to time. However, it would be very inconvenient to be bound to one unique gas station. One wants to select a gas station in one immediate vicinity for this purpose. This is a typical example of a call-based binding.
- **Per transaction binding**: The above two binding types actually are two ends of a spectrum of binding approaches. Transaction binding binds the object to a particular acquaintance object for the duration of a logical unit of interaction. The interaction may consist of multiple message sends and last for a considerable amount of time, but the complex of messages forms a logical entity.

4.2 Conditions for Binding

In the previous section, object binding was discussed, but not the way objects are selected to fulfil an acquaintance role. In traditional object-oriented systems, this is done based on matching the name of the object or by passing references. When passing references, the object has already been selected, so no further selection is required. However, the use of object references actually indicates that the name-based selection mechanism did not fit the requirements of the software engineer and therefore it was necessary to implement the selection mechanism on top of the language model, i.e. the selection of the object had to be hand-coded.

Object binding mechanisms can be categorised in several ways, but for the discussion in this paper we use the following classification:

- **Name-based binding**: The acquaintance is selected based on the name of the object. Although this approach may seem similar to conventional name-based binding, it actually is a generalisation. For example, when name-based binding is combined with per-call binding, the selected object may be different for every subsequent call to the acquaintance due to changes in the context of the calling object.
- **Class-based binding**: The object fulfilling the role of the acquaintance can also be selected based on the required class of the acquaintance. When binding the acquaintance, the selected object is located in the context of the object and is of the appropriate class. Combined name and class-based binding is used in strongly typed object-oriented languages such as C++.
- **Interface type based binding**: Languages such as Java support the notion of interfaces in addition to classes for the specification of types. Objects that (claim to) implement an interface can be used in situations where an object requires an acquaintance providing the specified interface.
- **Property-based binding**: When binding based on name, class and/or interface is not sufficient, the condition for binding can be defined as a property that an aspirant acquaintance object has to fulfil in order to be selected as an acquaintance. The property might be an arbitrary expression define the discriminating characteristic of the appropriate object. Since the name, class and interface type are properties of an object, the former binding approaches can be viewed as specialisations of property-based binding.

4.3 Number of Selected Objects

The acquaintances of an object need to be selected in the context of the object, for instance, by use of a broadcast mechanism. However, the result of the selection process might be that the set of appropriate objects consists of more than one object. Multiple objects in the context of the object are possible acquaintances. Traditionally, an object always sends a message to only a single receiver object. However, it might be the intention of the message send to address more than a single object. In any case, the appropriate number of objects representing an acquaintance needs to be specified. In this paper, three different types of object selection are used.

- **Single object**: Although the object selection process may result in multiple potential acquaintance objects, only one of these objects is selected for delivering the required service. Although one could imagine different approaches to select the acquaintance from the set of suitable objects, just selecting the first object is the simplest solution and does not result in any loss of generality. An example situation could simply be getting a cup of coffee. Although at a computer science department, in general, multiple coffee machines are available, the simplest solution is to select the closest one.

- **All objects**: The other extreme in the object selection process is to bind all suitable objects to the acquaintance and to multi-cast messages to the acquaintance to all these objects. A typical example is an emergency stop in a system that has to be broadcasted to all objects selected during the selection process.
- ***N* objects**: The compromising option is, of course, to select a subset of the potential acquaintance objects, i.e. to select *N* objects. This option can be used when an object wants to distribute a task to a finite number of objects, e.g. an image can be divided 16 squares, requiring 16 objects to be selected fro analysing the image squares.

4.4 Selection Region for Binding

The fourth dimension that plays a role in object communication is the region in which objects are searched for the object selection process. In large, distributed systems, the cost of a broadcast to all objects in the system often is incredibly high and the function cannot be used except for exceptional situations. Therefore, if an object wants to connect to its acquaintances, the region, i.e. the part of the object composition hierarchy, in which the potential acquaintances are searched has to be restricted. On the other hand, the object itself may require some acquaintance to be available locally.

- **Inside object**: The object can use its part objects as a selection set for dynamically linking an acquaintance. In this case, the object can, very flexibly, link one or more of its parts as an acquaintance object without having to distinguish between binding internal or external objects.
- **Immediate context**: As described in section 2, we assume a hierarchical object model, meaning that all objects (except the system itself) are encapsulated by another object *E*. A very logical region for selection then is the immediate context of the object consisting of all objects encapsulated by *E*.
- ***N* encapsulating contexts**: The selection region can be extended to more than only the immediate context. This option indicates that the region for selection is enlarged to ***N*** encapsulating contexts. For example, in the case that ***N***=2, the context would consist of all objects in the same context and all objects in the immediate context of the encapsulating object. This option is related to the issues discussed in section 4.3. For instance, in case of a selecting a single object, the search may begin in the immediate context of the object and subsequently continue until a suitable acquaintance is found or the maximum of ***N*** encapsulating contexts is reached.
- **Global**: The final option, *global*, uses the complete system as the selection region, i.e. starting from the immediate context of the object the search for suitable acquaintance objects is performed until sufficient suitable candidates have been found. Since the search starts in the immediate context of the object, the *global* search is not the same as a system-wide broadcast and is commonly used when selecting a single or a smaller number of acquaintance objects.

5 Layered Object Model

The layered object model (`LayOM`) is our research language model that has successfully been applied to several problems associated with the traditional object-oriented paradigm. In this paper we discuss our solution to the acquaintance handling problem in the form of the `Acquaintance` layer type as a part of the layered object model. In the remainder of this section, first the basics of the layered object model are introduced. Subsequently, in section 5.2, some examples of the extended expressiveness of `LayOM` are discussed. The `Acquaintance` layer type is introduced in section 5.3 and some examples of its use are presented in section 5.4. Finally, the implementation `LayOM` and the supported layer types is discussed in section 5.5.

5.1 LayOM

The layered object model is an extended object model, i.e. it defines in addition to the traditional object model components, additional components such as layers, states and categories. In figure 1, an example `LayOM` object is presented. The layers encapsulate the object, so that messages send to or by the object have to pass the layers. Each layer, when it intercepts a message, converts the message into a passive message object and evaluates the contents to determine the appropriate course of action. Layers can be used for various types of functionality. Layer classes have, among others,

been defined for the representation of relations between classes and objects, design patterns and programming conventions. In this paper, the representation of object communication through the use of layers is discussed.

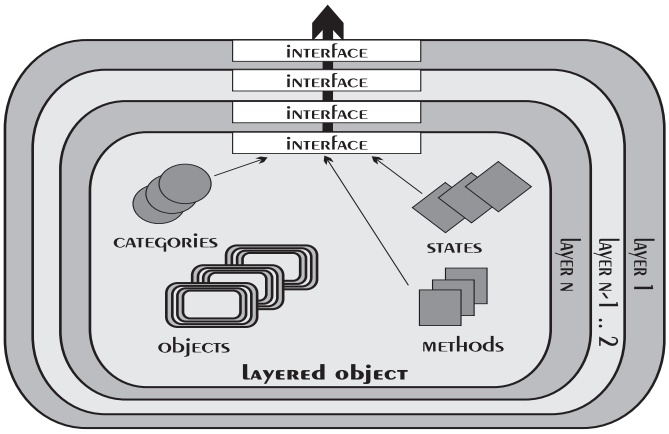

Figure 1. The layered object model

A **LayOM** object contains, as any object model, instance variables and methods. The semantics of these components is very similar to the conventional object model. The only difference is that instance variables, as these are normal objects, can have encapsulating layers adding functionality to the instance variable.

A *state* in **LayOM** is an abstraction of the internal state of the object. In **LayOM**, the internal state of an object is referred to as the *concrete state*. Based on the object's concrete state, the software engineer can define an externally visible abstraction of the concrete state, referred to as the abstract state of an object. The abstract object state is generally simpler in both the number of dimensions, as well as in the domains of the state dimensions. We refer to [Bosch 94b] for an extended description of abstract object state.

A category is an expression that defines a set of objects that are treated similarly by the object. This set of objects treated as equivalent by the object we denote as *acquaintances*. A category describes the discriminating characteristics of a subset of the external objects that should be treated equally by the class. The behavioural layer types use categories to determine whether the sender of a message is a member of an acquaintance category. If the sender is a member, the message is subject to the semantics of the specification of the behavioural layer type instance.

A layer, as mentioned, encapsulates the object and intercepts messages. It can perform all kinds of behaviour, either in response to a message or pro-actively. Previously, layers have primarily been used to represent relations between objects and for the representation of design patterns. In **LayOM**, relations have been classified into structural relations, behavioural relations and application-domain relations. *Structural relation* types define the structure of a class and provide *reuse*. This relation type can be used to *extend* the functionality of a class. The second type of relations are the *behavioural relations* that are used to relate an object to its clients. The functionality of the class is used by client objects and the class can define a behavioural relation with each client (or client category). Behavioural relations *restrict* the behaviour of the class. For instance, some methods might be restricted to certain clients or in specific situations. The third type of relations are *application domain relations*. Many domains have, next to reusable application domain classes, also application domain relation types that can be reused. For instance, the *controls* relation type is a very important type of relation in the domain of process control. In the following section, structural and behavioural relation layer types will be discussed in more detail. For information on application-domain relation types, we refer to [Bosch 95b].

Next to being an extended object model, the layered object model also is an *extensible* object model, i.e. the object model can be extended by the software engineer with new components. **LayOM** can, for example, be extended with new layer types, but also with new structural components, such as *events*. The notion of extensibility, which is a core feature of the object-oriented paradigm, has been applied to the object model itself. Object model extensibility may seem useful in theory, but in order to apply it in practice it requires extensibility of the translator or compiler associated with the language. In the case of **LayOM**, classes and applications are translated into C++. The generated classes can be combined with existing, hand-written C++ code to form an executable. The **LayOM** compiler is based on *delegating compiler objects* [Bosch 95a], a concept that facilitates modularisation and reuse of compiler specifications and extensibility of the resulting compiler. The implementation of the **LayOM** compiler is discussed in section 5.5.

5.2 Relations and Design Patterns

The primary novel feature of the layered object model are the layers that encapsulate **LayOM** objects. Therefore, we discuss, in this section, several examples of layer types that have been defined earlier. The discussion in this section forms an introduction and context for the *Acquaintance* layer type discussed in section 5.3.

5.2.1 Structural Relations

Structural relation types, as described above, define the *structure* of an application. A class uses the structural relations to *extend* its behaviour and the class can be seen as the *client*, i.e. the class that obtains functionality provided by other classes. Generally, three types of structural relations are used in object-oriented systems development: *inheritance*, *delegation* and *part-of*. These types of relation all provide some form of *reuse*. The inherited, delegated or part object provides behaviour that is reused by, respectively, the inheriting, delegating or whole object. Therefore, next to referring to these relation types as *structural*, we can also define them as *reuse* relations.

Orthogonal to the discussed relation types one can recognise two additional dimensions of describing the extended behaviour of an object, i.e. *conditionality*, and *partiality*. Conditionality indicates that the reusing object limits the reuse to only occur when it is in certain states. Partially indicates that the reusing object reuses only part of the reused object.

In **LayOM**, a relation between two classes or objects, regardless of its complexity, is modelled as a single entity within the model. An alternative approach would be to define a collection of orthogonal constructs and to decompose a relation between objects into instances of each orthogonal construct as is done in. This approach does, however, not represent a conceptual entity in analysis and design as an entity in the language model.

The different aspects of a structural relation, i.e. its type, partiality and conditionality, form a three-dimensional space which contains all possible combinations. In compliance with our modelling principle, we define a relation type for each combination. This results in twelve structural relation types. These structural relation types are shown in table 2.

relation type	**inheritance**	**delegation**	**part-of**
default	Inherit	Delegate	PartOf
conditionality	ConditionalInherit	ConditionalDelegate	ConditionalPartOf
partiality	PartialInherit	PartialDelegate	PartialPartOf
conditional and partial	CondPartInherit	CondPartDelegate	CondPartPartOf

Table 2. Structural relation types

Due to space constraints, it is not possible to describe the syntax and semantics of all structural relation types. Instead, one layer of type *Partial Inheritance* is described in detail. The semantics of the other layer types can be deduced relatively easy, but for an extensive description of the semantics of the other structural relation types we refer to [Bosch 94a, Bosch 95b].

An example class `TextEditWindow` contains a partial inheritance layer with the following configuration:

```
pin: PartialInherit(Window, *, (moveOrigin));
```

The semantics of the partial inheritance layer are that a part of the interface of the inherited class is reused or excluded. The name of this layer is `pin` and its type is `PartialInherit`. The layer type accepts three arguments. The first argument is the name of the class that is inherited from; `Transformer` in this case. The second argument is a '*' or a list of interface elements and indicates the interface elements that are to be inherited. The '*' in this example indicates that all interface elements are inherited. The third argument can also contain a '*' or a list of interface elements. In this

example, the list only consists of one element: `moveOrigin`. The semantics of layer `pin` is that class `TextEditWindow` inherits the complete interface of class `Window`, except for `moveOrigin`.

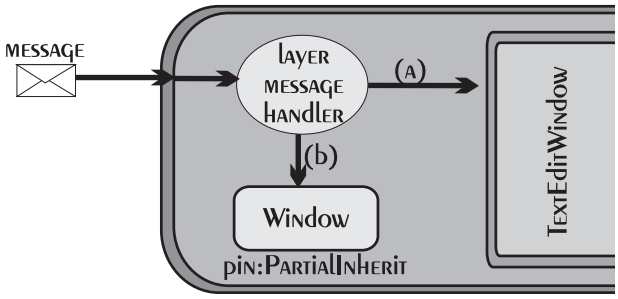

Figure 2. The PartialInherit **layer**

In figure 2, the implementation of this semantics is illustrated. The partial inheritance layer is the second layer of class `TextEditWindow`. There is the most outer layer, shown around layer `pin` and an inner layer, shown around `TextEditWindow`. All inheritance layers create an instance of the inherited superclass. In figure 2, an instance of class `Window` is shown. The layer contains a message handler, that, for each received message, determines whether the message is passed on inwards or outwards or that it is redirected to the instance of class `Window`. In the figure, an incoming message is shown. The message is reified and handed to the message handler. The message handler will read the selector field of the message and compare it with the (partially) inherited interface of class `Window`. If the selector is part of the set of interface elements, the message is redirected to the instance of class `Window` (situation (b) in figure 2). If the selector does not match with the interface of class `Window`, it is not redirected but forwarded to the next layer (situation (a) in figure 2).

5.2.2 Behavioural Relation Layers

In the previous section, we discussed relations where the object containing the relation is the client, i.e. the object 'extends' itself with the structural relations. In this section we discuss relation types between the object and its clients. These relation types, generally, constrain the access of clients and the behaviour of the object in some way.

In order to keep the type and number of clients of the object open ended, **LayOM** makes use of categories and for each message is determined whether the sender of the message is a member of a category. The message is then subject to the behavioural relation(s) defined for that client category.

A relation between the object and a client category can be defined by a number of behavioural constraints. The types of constraints are *client-based access*, *state-based access*, *concurrency* and *real-time*. The layered object model takes a different approach to defining modelling constructs when compared to the conventional approaches. Rather than aiming at defining 'clean', orthogonal constructs, we try to define constructs that correspond to conceptual entities. Hence, we are convinced, supported by the object-oriented analysis and design methods, that the concept of a relation between objects is a conceptual entity and that the different aspects of this relation should be defined as part of this relation, rather than as several unrelated orthogonal constructs that, when combined, provide equivalent behaviour.

In table 3, the behavioural relation (or layer) types are shown. Each relation type can be viewed as a location in the space build by the four constraint dimensions defined above. However, a relation always requires a client category to be specified. This results in three dimensions which can be part of or not part of the relation. For each combination, a relation type is defined.

No factor	**One factor**	**Two factors**	**Three factors**
RestrictClient	RestrictState	RestrictStateAndConc	RestrictStateConcAndTime
	RestrictConc	RestrictStateAndTime	
	RestrictTime	RestrictConcAndTime	

Table 3. Behavioural relation types

Again, due to space constraints, we are unable to discuss the semantics of all behavioural relation types. Instead, we will discuss the semantics of the `RestrictState` layer type in detail.

The `RestrictState` layer type restricts the access for a particular client category when the object is in a certain states. Class `TextEditWindow` has a `RestrictState` layer with the following configuration syntax:

```
rs : RestrictState(Programmer, accept all when distFromOrigin < 100 otherwise reject);
```

The semantics of this layer specification is the following. Clients that are classified as members of the `Programmer` client category can access all methods of the object provided that `distFromOrigin` is less than or equal to 100. If it is larger than 100, the message is rejected, i.e. an error message is returned to the client object that sent the message.

In figure 3, the functionality of the `RestrictState` layer type is presented graphically. The layer will intercept messages sent to the object. If the message is sent by an object that is not classified as a member of the client category that is indicated by the layer specification, the message will just be passed on to the next layer. Otherwise, the selector of the message is matched with the identifier list in the layer specification. If it matches, the state that is referred to in the specification is evaluated. Depending on the use of the keyword `unless` or `when`, the message will be passed on to the next layer (see (a) in figure 3). If the message is not passed on, the message is stored in the message queue if the `delay` keyword is used (see (b) in figure 3) or, in case of the use of keyword `reject`, the message is discarded and an error message is sent to the sender object (see (c) in figure 3).

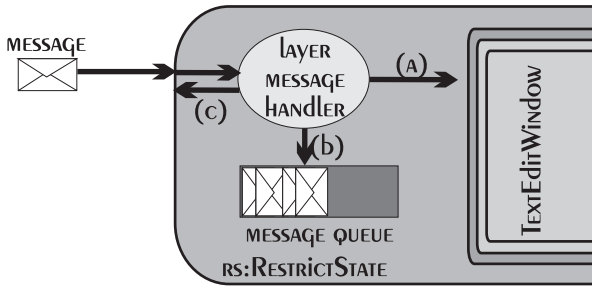

Figure 3. The RestrictState **layer**

5.2.3 Design Patterns

Design patterns are becoming increasingly popular as a mechanism to describe solutions to general design problems that can be customised to particular applications. Most design patterns have a well-defined semantics that can be used as the basis for defining language constructs that explicitly support the representation of the design pattern in the programming language. We consider design patterns as a very important technique used during design. However, we have identified problems associated with the implementation of patterns in the conventional object-oriented languages. These languages, like C++, provide insufficient support to implement design patterns in a traceable, efficient manner. Traceability is problematic because several design patterns are lost during implementation. In addition, some design patterns require the definition of a, potentially large, number of methods with very trivial behaviour, e.g. forwarding a message to a nested object. The latter is inefficient from the perspective of the software engineer.

In **LayOM**, layer types have been defined that represent design patterns. The design patterns have been selected from the collection defined by [Gamma *et al.* 94], in particular from the structural and behavioural design patterns. Since the collection contains seven structural design patterns and 11 behavioural design patterns, we are unable to describe layer types for all these design patterns for reasons of space. Instead we limit ourselves to describing the *Facade* pattern. We refer to [Bosch 96a] for a more complete discussion on layer types for design pattern representation.

The *Facade* design pattern is used to provide a single, integrated interface to a set of interfaces in a subsystem. `Facade` defines a higher-level interface that simplifies the use of the subsystem. The function of the subsystem (`Facade`) class is twofold, i.e. coordination between the classes in the subsystem and providing an integrated interface to clients of the subsystem. However, the forwarding of messages to objects within the subsystem is problematic. The traditional approach is to define a method in the subsystem class that forwards a message sent by a client to the appropriate object inside the subsystem. The disadvantage of this approach is that for every message send by a client the subsystem class has to define a method. The number of methods might easily grow very large and defining these methods is much work for just forwarding a message. A second disadvantage is that the traceability of the design pattern in the implementation is lost.

As a solution to the identified problems associated with the traditional implementation approach one can, within the layered object model, make use of the Facade layer type. The Facade layer type provides the functionality of forwarding messages to objects that are part of the subsystem. A layer of type Facade is defined as follows:

```
<id> : Facade(forward <mess-sel>+ to <object>, forward <mess-sel>+ to <object>, ...);
```

The behaviour of a Facade layer is the following. It matches the selector of the message with the message selectors defined in <mess-sel>+, the message will be forwarded to the object <object>. The Facade layer can contain several forwarding elements, allowing the subsystem class to forward to several objects using a single Facade layer. A subsystem class using a Facade layer can be defined as follows:

```
class SubSystemA
   layers
      face : Facade(forward mess1, mess2 to PartO1, forward mess3 to PartO2);
      PartO1 : PartOf(ClassOfO1);
      PartO2 : PartOf(ClassOfO2);
   ...
end; // class SubSystemA
```

5.3 Acquaintance Layers

Almost any object, in the course of its operation, communicates with other objects for achieving its own goals. We refer to the objects that an object needs to communicate with as *acquaintances*. An acquaintance may provide some services to the object, it might request services from the object or it may both request and provide services. In object-oriented systems, the amount of work required from the software engineer to connect the various objects should be as little as possible. The object itself should contain the specifications on how to connect to its acquaintances and what requirements an potential acquaintance should fulfil. On the other hand, when an object is used in an unanticipated context, the software engineer requires expressive and modular language constructs to connect an object to the other objects.

As we discussed in section 3, one can identify a number of problems associated with the way traditional object-oriented languages deal with acquaintance handling, among others, related to reusability and expressiveness. This means that other approaches have to be developed to address these problems. In section 4, those aspects of acquaintance handling were discussed that we believe to be relevant. In this section, we introduce a new layer type, Acquaintance, that provides a solution to the identified problems in the context of LayOM. The work on the Acquaintance layer type has three goals:

- The *expressiveness* of the acquaintance handling mechanisms that are part of the language should be improved so that the software engineer less often has to resort to hand-coding acquaintance selection and binding through explicit pointer.
- The requirements of an object on its acquaintances should be *separated* from specification of the selection and binding of the acquaintances.
- The inter-twining of code implementing the functionality of a method with the code used for acquaintance handling should be removed.

Within the context of the layered object model, introduced in the previous section, we have defined a new layer of type Acquaintance. The syntax of the layer type is as follows:

```
<layer-id>:   Acquaintance([<obj-name>|<category>] is-bound [permanent|per-call|from <selector>
              until <selector>] for [one|all|<n>] objects from [inside|<n> contexts|global]);
```

An instance of this layer type is bound based on the name of a called object or based on the name of a category. The actual object that is communicated with may have been bound permanently on object instantiation, bound for every call or during a transaction starting with a <selector> and ending with the same or another <selector>. When selecting the appropriate object to bind the acquaintance to, the search may be inside the object itself, in <n> contexts of the object or globally. The notion of *contexts* is important in our underlying system view where the objects in the system are organised hierarchically. The first context of the object consists of those objects that are located in the immediate vicinity of the object, i.e. in the same subsystem, etc.

Layers can be defined in the class specification, as is often done for structural relations. However, it is also possible to define layers for individual instances, i.e. an instance is created from a class and subsequently extended with one or more layers. Thus, the software engineer can define a class as a component and include layer specifications for all acquaintances of the object. Alternatively, one can define the class and only declare the required acquaintances. When composing a system using that class, the software engineer can add layer specifications for selecting and binding acquaintances. The latter allows one to define the communicating behaviour of the object independently of the functionality of the object. However, one should note that the Acquaintance layer only specifies the selection and binding process. It contains no functionality for specifying the properties required by the object of the acquaintance. The categories are used to specify the properties an acquaintance must fulfil, whereas the Acquaintance layer specifies how to select and bind a candidate acquaintance from the context. Note that, in the situation where the acquaintance only has to have a particular name, instead of the category the object name can be used in the Acquaintance layer and no category has to be defined. With respect to the various acquaintance handling aspects discussed in section 4, one can conclude that all mentioned aspects are supported and implemented by the Acquaintance layer type.

To illustrate this in the layered object model, we discuss an example from mobile telephony. A physical mobile phone is represented by a mobile phone object in the distributed information system of a mobile telephony operator. A mobile phone object has a number of acquaintances, i.e. a base station object, an operator object and a voice mail object. Since the owner of the physical phone moves around, the phone object also moves through the distributed information system. In some cases, the phone user travels outside the region covered by the operator, in which case another operator that covers the current region provides telephone services for the phone, under the condition that the two operators have such an agreement. Below, part of the **LayOM** specification for the mobile phone class is shown.

```
class MobilePhone
   layers
      base : Acquaintance(base-station is-bound per-call for one object from 1 contexts);
      oper : Acquaintance(operator is-bound from connect-operator until change-operator
               for one object from global);
      mail : Acquaintance(voice-mail is-bound permanent for one object from global);
      ...
   categories
      base-station begin acq.subClassOf(BaseStation) and acq.owner = operator; end;
      operator begin self.operator = acq or acq.relatedOperator(self.operator); end;
      voice-mail begin acq.subClassOf(VoiceMail) and acq.phoneNumber = self.phoneNumber; end;
   ...
end;
```

The specification only contains the layer and category definitions for the aforementioned acquaintances of the mobile phone object. The categories describe the discriminating characteristics of the acquaintance objects, whereas the layers describe how to bind actual objects to each acquaintance. The object name acq used in the category specifications is a pseudo variable used to refer to the acquaintance being defined. Note that one could have defined the MobilePhone class without the Acquaintance layers. That allows one to add the layer specifications to individual instances of the MobilePhone class, thus creating unique instances.

In this section, we have introduced the layered object model and shown how **LayOM** objects can be used as components in a component-based system. Each component has acquaintances that it communicates with during its lifetime, but the specification of the selection of and communication with the acquaintances is not well supported in the conventional object model and has to be implemented on top of the object model. The layered object model allows for the separation of the functionality of the object and the way it selects its acquaintances, thereby, among others, increasing the reusability and understandability.

5.4 Examples of Acquaintance Handling

In this section, several examples of acquaintance selection and binding using acquaintance layers are presented. These examples capture the main expressiveness available through the acquaintance layer type.

5.4.1 Per-call acquaintance binding

Earlier in the paper, the example of a class Product was presented. An instance of this class represents a product under production in an integrated manufacturing system. During production, several production steps have to per-

formed on the product. The various production steps are provided by different production machines. This requires that a product transported between the various production machines. In a fully automated manufacturing system, transport could be provided by so-called automatic guided vehicles (AGVs). In this example, selecting and using an AGV is shown

```
class Product
   layers
      transport : Acquaintance( transporter is-bound per-call for one object from global);
      ...
   categories
      transporter begin acq.instanceOf(AutomaticGuidedVehicle) end;
      ...
   methods
      moveTo(Position newPos)
         begin
            transporter.transport(self.currentPosition, newPos);
         end;
   ...
end; //class Product
```

The product can, obviously, not bind itself to a single AGV instance, since this would be too rigid. Instead, an AGV is selected when the need for transportation arises. Internally in the product, an acquaintance `transporter` is used to refer to entities that can transport the product. The category `transporter` is used to define the requirements candidates objects need to fulfil, i.e. be an instance of class `AutomaticGuidedVehicle` or one of its sub-types.

When the object sends a message to its acquaintance transporter, a normal message is created and sent. Since, within the product object no object with the name transporter exists, the message is sent on outward and starts passing layers of the object. When the message reaches the `Acquaintance` layer with the name `transport`, the layer will match the receiver name in the message with the category name stored in the layer specification. Since the two match, the layer will delay the message since it first needs to find an appropriate object that can fulfil the role of the acquaintance in the context of the object. The layer searches for objects that fulfil the requirement specified in the associated category. Since it only needs one object, it stops the search as soon as it finds a matching object, i.e. an AGV. The receiver field in the intercepted message is assigned the identity of the selected object and the message is directed to that object. Since the binding of the acquaintance is performed on a per-call basis, the identity of the selected object does not need to be stored by the layer, as would be the case for more permanent binding techniques.

5.4.2 Per-transaction binding

In the previous section, per-call binding of an acquaintance was used. This was done to select the most appropriate candidate in the context of the object. Although this is a very flexible form of binding, sometimes an object needs to bind to a particular acquaintance for a longer time period than a single message send. The object may need to bind itself for the period of a transaction that starts with a particular action, followed by a sequence of operations that are part of the transaction and then closed by a closing action. However, the use of an acquaintance inside the object itself should not depend on the type of binding and all functionality associated to transaction-based binding should be confined to the acquaintance layer.

The example that we use here to illustrate transaction-based binding is a mobile phone. This particular mobile phone is still rather primitive and must remain in contact with the base station via which a telephone call was initiated for the complete duration of the call. Below part of the code for the mobile phone class is shown.

```
class MobilePhone
   layers
      base : Acquaintance(base-station is-bound from openCall until closeCall for one object
             from 1 contexts);
      ...
   categories
      base-station begin acq.subClassOf(BaseStation) and acq.owner = self.operator; end;
      ...
   methods
      performCall(pn : Phonenumber)
         begin
            cd : CallData;
```

```
            cd := base-station.openCall(pn);
            while cd.ongoing and not self.callEndRequest do
               begin
                  self.sendFrame;
                  self.receiveFrame;
               end;
            base-station.closeCall(cd);
         end;
      ...
end; // class MobilePhone
```

Within the mobile phone object, the name `base-station` is used as an ordinary object name. When the object sends a message to the acquaintance, the message is intercepted and stopped by the acquaintance layer. If the acquaintance has been already been bound for a transaction, the message is forwarded to the selected object. If the message selector matches with the transaction ending selector specified in the layer specification, in this case `closeCall`, the reference to the bound object is discarded and the state of the layer adjusted. If, according to the layer, no transaction is ongoing, a search process similar as described in the previous example will take place. Note that an appropriate acquaintance, in this example, not only has to be of a particular type, but also should fulfil certain state requirements, i.e. the base station should be owned by the operator servicing the mobile phone.

5.4.3 Multiple objects

In the two previous examples, the acquaintance used by the object was bound to a single external object. This, however, is not a requirement. An acquaintance can be bound to the set of objects that fulfil the requirements specified in the category representing the acquaintance. A message by the object to the acquaintance will be multi-casted to all objects in the set of appropriate objects. Note that this is symmetrical to the behavioural relation layers discussed in section 5.2.2. All clients of the object that are members of a category are subject to the behavioural relation defined by the object.

The example we use to illustrate the use of a multi-casting acquaintance layer is in the domain of electricity distribution. As described, in future house holds all electric appliances will contain a microprocessor and will communicate over the 240V power lines. This results in a distributed system of objects, where objects represented real-world entities such as lamps, switches, televisions, etc. Design of such systems often contain a manager object that takes care of certain tasks in the system. One task is stopping all usage of electric power in the house and this task becomes relevant, among others, in emergency situations.

```
class PowerManager
   layers
      el-users : Acquaintance(power-user is-bound per-call for all objects from 1 contexts);
      ...
   categories
      power-user begin acq.powerUsage > 0; end;
   ...
   methods
      emergencyStop
         begin
            power-user.maxPowerUsage(0);
         end;
   ...
end; // class PowerManager
```

In the class `PowerManager` shown above, a method `emergencyStop` is defined that forces all entities currently consuming power in the house to restrict their power usage to zero. Inside the object, only an acquaintance with the name `power-user` is used and a message is sent to it. The acquaintance layer intercepts the message and selects all objects that fulfil the requirement specified in the category. Note that searching is restricted to the immediate context of the power manager object. Since all selected objects are bound to the acquaintance, the message sent by the object is multi-casted to all these objects.

5.5 Implementation

As mentioned earlier, the layered object model is both an *extended* and an *extensible* object model. Extended, since it contains components in addition to the components of the traditional object model, i.e. states, categories and layers. Extensible, since it may be extended with new components whenever the software engineer considers it to be appropriate. Possible extensions are new layer types, but also new object components such as events. Over the years, primarily new layer types for the representation of structural and behavioural inter-object relations and design patterns have been defined. In this paper, we have discussed a layer type for acquaintance handling which provides expressive constructs for acquaintance selection and binding. However, over time it may become clear that the proposed layer type for acquaintance handling lacks expressiveness for particular situations or should be implemented differently. In response to this, the language should be extended or changed to reflect these new requirements. In conventional object-oriented languages this, would be impossible due to the monolithic, rigid compiler technology that is used to construct compilers for these languages. However, it is important to make clear that the rigidness of conventional programming languages is due to technological problems, i.e. compiler technology, and not due to conceptual or other human factors. The software engineer, while modelling a domain, ideally is able to represent each domain concept with a corresponding concept in the programming language.

Since traditional compiler technology is unsuitable to implement extensible languages, an alternative approach had to be defined. The implementation of the layered object model is based on the notion of *delegating compiler objects* (DCOs) [Bosch 95a]. A DCO is an object that compiles a part of the syntax of the input language. It consists of one or more lexers, one or more parsers and a parse graph. The nodes in the parse graph have the ability to generate code for themselves. In case of the `LayOM` class compiler, it consists of a class DCO, method DCO, state DCO, category DCO and a DCO for each layer type. Each DCO definition results in a class and a DCO object can instantiate another DCO and delegate control to it. The delegated DCO will perform its functionality and return control to the delegating DCO when it is finished. The `LayOM` compiler DCOs generate C++ output code. `LayOM` code is either a class or an application. A `LayOM` class is compiled into a C++ class and a `LayOM` application is compiled into a C++ main program. The generated C++ class can be incorporated in any C++ function and, subsequently, into an executable program.

An advantage of using the DCO approach is that it supports extensibility of the language very well. When the software engineer wants to add a new concept to the language, all that is required is a DCO defining the syntax and code generation information of that particular concept. The new DCO is added to the set of DCOs in the existing compiler and it can be used immediately. Except for some minor modifications in the DCOs that should instantiate the new DCO, no changes are required. For a more detailed description of DCOs, we refer to [Bosch 95a].

6 Related Work

The domain of object communication has been studied by many authors. Since message passing forms the basis of the object-oriented paradigm, this is not very surprising. However, one can recognise two main topics that people have been trying to address. The first is the communication between objects in the presence of distribution. Here, the traditional object-communication is used as a basis, but its domain of operation is extended over individual address spaces. Most prominent is the work on CORBA [OMG 96], OLE/COM [Microsoft 96], etc. Their relation to this work is primarily that of enabling technology. Without means to cross address spaces, the concepts and techniques described in this paper are less interesting and valuable.

The second topic that has been studied is that of *object group communication*. Several authors have identified that the traditional means of communication between a sender and a known receiver is too restrictive. Although object group communication is not supported in the traditional object-oriented model, solutions and extensions have been proposed by several authors. Most authors are concerned with representing complex communication, or interaction, patterns between a group of objects where each object plays some roles in the pattern. The work on *contracts* [Helm et al. 90] presents an extension to the conventional object-oriented paradigm that is capable of representing such interaction patterns and some formal aspects of such patterns. *Abstract communication types* (ACTs) [Aksit et al. 93] is a similar approach that uses message reflection to implement interaction patterns. Gluons [Pintado 95] also represent interaction patterns, but contain a finite state machine to describe the interaction protocol. All these approaches are concerned with representing an interaction pattern between a group of objects as a separate entity that interacts with the involved objects. This is different from what we are concerned with in this paper; it is our aim to represent the acquaintance selection and binding process of an individual object in an expressive and reusable manner.

A domain that has studied object selection extensively is object-oriented database (OODB) technology, e.g. GemStone [Breitl et al. 89] and ObjectStore [ObjectStore 93]. OODBs generally are primarily concerned with object persistence and the persistence of object references, but it is possible in an OODB to select an object from a collection through querying the collection and use the selected object as an acquaintance. The object selection techniques described in this paper bare a strong resemblance with querying techniques in object-oriented databases. Although efficiency was not one of our primary goals, we believe that incorporating advanced querying mechanisms from the object-oriented database domain will considerably improve the performance of acquaintance selection. Despite this correspondence, acquaintance handling and OODBs are obviously quite different concepts. Also, since the query statements often are embedded in the method code many of the problems discussed in section 3 are also valid for object-oriented databases.

7 Conclusion

In this paper we have studied communication between objects and the way objects deal with acquaintances. Although the traditional means of object communication might be rather satisfactory in small, rigid applications, large, distributed and flexible object-oriented systems have, at least, four characteristics that complicate object communication. First, the system often is distributed which requires some notion of object hierarchy. Second, it contains large numbers, e.g. thousands, of objects, requiring a more advanced organisation than the 'sea-of-object' approach. Third, objects need to be relocated at run-time and, fourth, objects can be replaced by other objects in order to adapt to the dynamic changes in the system. These characteristics require acquaintance selection and binding to be flexible at run-time.

Traditional object communication is based on sending a message to a receiver object known to the sender of the message. At instantiation time, an object establishes its acquaintances through naming and uses these objects through its life time. If this is too rigid, the software engineer has to implement the binding of objects manually using pointers. In our experiments we found the traditional communication semantics too limited and we identified several problems. First, object reusability is reduced since the object cannot be reused in situation where an acquaintance should be selected and bound in a different way. Secondly, the acquaintance selection and binding mechanism cannot be reused separately from the object in which it is defined. Thirdly, the understandability of the code is reduced since code representing the behaviour of the object is mixed with code for handling acquaintances. Finally, one can identify a considerable lack of expressiveness for dealing with object acquaintances and more expressive constructs should be investigated.

In an attempt to provide a solution to these problems, we have studied the expressiveness one would like to see supported for acquaintance handling. We identified four important aspects that we believe should be supported. The first of these aspects is the type and duration of acquaintance binding, i.e. an acquaintance can be bound permanently, for every call or for the duration of a transaction. The second aspect is the condition used for selecting the acquaintance. This can be based on the name or the class of the acquaintance or based on another arbitrary property. Thirdly, since candidate acquaintances are selected based on properties, more than one object may be selected. Therefore, one may either bind a single object, all selected objects or a subset. Finally, the region in which is searched for candidate acquaintances is relevant. One can use the objects contained in the object as a selection set, the objects in the immediate context of the selecting object, the objects in ***N*** encapsulating contexts or the search may take place globally, i.e. throughout the complete system.

Based on the above analysis, we have recognised that is important to separate an class or object's requirements on its acquaintances from the way an object selects and binds its acquaintances in actual systems. The model we propose as a solution makes use the use of *acquaintance layers* as part of the *layered object model* (`LayOM`). Acquaintance layers uniformly extend the traditional object-oriented communication semantics and allow for the first-class representation of acquaintance handling specifications. The acquaintance layers provide the expressiveness for the four aforementioned aspects of acquaintance selection and binding. The acquaintance layers solve the identified problems and increase object reusability. The layer mechanism in `LayOM` allows for selection mechanism reusability since the software engineer can define proprietary layer types for acquaintance handling. Understandability of the code is improved since, in the method, an acquaintance can be used as a normal object and all selection and binding is performed in the acquaintance layer. Finally, the expressiveness of the acquaintance layer for acquaintance handling is considerably better than the that of the traditional object model.

Acknowledgements

The discussions with Fredrik Ygge and the detailed comments from Peter Molin helped develop the ideas presented in this paper.

References

[Aksit et al. 93]. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, 'Abstracting Object Interactions Using Composition-Filters,' in *Object-based Distributed Programming*, R. Guerraoui, O. Nierstratz, M. Riveill (eds.), LNCS 791, Springer-Verlag, 1994.

[Bosch 95a]. J. Bosch, 'Parser Delegation - An Object-Oriented Approach to Parsing,' in *Proceedings of TOOLS Europe '95*, pp. 55-68, 1995.

[Bosch 95b]. J. Bosch, 'Relations as Object Model Components,' to be published in the *Journal of Programming Languages*, 1995.

[Bosch 95c]. J. Bosch, 'Layered Object Model - Investigating Paradigm Extensibility,' *Ph.D. dissertation*, Department of Computer Science, Lund University, November 1995.

[Bosch 96a]. J. Bosch,'Language Support for Design Patterns,' in proceedings of *TOOLS Europe '96*, pp. 197-210, 1996.

[Bosch 96b]. J. Bosch, 'Delegating Compiler Objects - An Object-Oriented Approach to Crafting Compilers,' in *Proceedings Compiler Construction '96*, 1996.

[Breitl et al. 89]. R. Breitl, The GemStone data management system. In *Object-Oriented Concepts, Databases and Applications*, W. Kim, F. Lochovsky (eds.), pp. 283-308, Addison-Wesly, 1989.

[Edelson 92]. D.R. Edelson, 'Smart Pointers: They're Smart, But They're Not Pointers,' *Proceedings USENIX C++ Conference*, pp. 1-19, 1992.

[Enersearch 96]. The Information, Society, Energy & System (ISES) project conference documentation, *Enersearch AB*, Sweden, August 1996.

[Francez et al. 86]. N. Francez, B. Hailpern, G. Taubenfeld, 'Script: A Communication Abstraction Mechanism and its Verification,' *Science of Computer Programming*, 6, 1, pp. 35-88, 1986.

[Gamma et al. 94]. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, October 1994.

[Helm *et al.* 90]. R. Helm, I. Holland, D. Gangopadhyay, 'Contracts: Specifying Behavioral Compositions in Object-Oriented Systems,' *Proceedings OOPSLA'90*, pp. 169-180, 1990.

[Microsoft 96]. The Microsoft Object Technology Strategy: Component Software, *Microsoft*, Part No. 098-55163, February 1996.

[Nierstrasz & Tsichritzis 95]. O. Nierstrasz, D. Tsichritzis, *Object-Oriented Software Composition*, Prentice-Hall, 1995.

[ObjectStore 93]. ObjectStore Release 3.0 Documentation, *Object Design, Inc.* Burlington, 1993.

[OMG 96]. The Common Object Request Broker: Architecture and Specification, Revision 2.0, *Object Management Group*, July 1996.

[Pintado 95]. X. Pintado, 'Gluons and the Cooperation between Software Components,' in [Nierstrasz & Tsichritzis 95], pp. 321-349, 1995.