

A Middleware-Oriented Visual Notation for Distributed Mobile Systems*

G. Reggio, M. Cerioli, and E. Astesiano

DISI, Università di Genova - Italy
{reggio,cerioli,astes}@disi.unige.it

Abstract. We are witnessing a growing demand for applications characterized by decentralization and mobility for which the peer-to-peer paradigm seems to have some clear advantages. The use of a middleware, encapsulating the treatment of distribution and mobility and providing appropriate abstractions for handling them in a transparent way, improves and simplifies the development of such applications.

In this paper we present a visual notation supporting the development of software based on such middleware. Our notation is characterized by the integration of the middleware into an OO paradigm, as an object whose operations correspond to the middleware primitives, and is UML-based, in the sense that, following the official UML [12] terminology, we propose a new UML profile. The models in our profile include diagrams to describe the software and the data on each kind of peer, the required cooperations among peers, the architecture of the overall system and the deployment on an actual network.

Introduction

Motivations The role and the importance of the middleware in the development of distributed systems is now well recognized (J. Charles, [2]). Indeed “middleware enables application engineers to abstract from the implementation of low-level details” (W. Emmerich in [4]). However there is a danger in the current success of the use of middleware in the industry; namely, the direct use of middleware products at the programming level without an appropriate software engineering support for the other development phases. On the other hand, as it is argued in [4], the software engineering support, at the design level at least, has to take middleware explicitly into account. Thus, “the software engineering community needs to develop middleware-oriented design notations, methods and tools ...”.

The main aim of this paper is to show how well-established software engineering design techniques can be tailored to provide support for middleware-oriented design.

We propose a visual object-oriented notation to support the development of software for a mobile distributed environment. Due to its large diffusion, and to

* Partially supported by Murst - Programma di Ricerca Scientifica di Rilevante Interesse Nazionale Saladin.

its visual and object-oriented nature, we base our notation on UML. Technically, we will present such notation as a UML profile [8]. A profile provides some mechanisms for specializing its reference metamodel in specific domains, e.g., to handle real-time systems. Our profile will be targeted to build applications using peer-to-peer middleware. Indeed, our work has originated from a national project (Saladin) aimed at providing support for distributed and mobile applications. In that area the peer-to-peer paradigm seems to have some clear advantages over some more traditional approaches, such as the client-server one. The very recent appearance in the market of peer-to-peer middlewares by commercial giants (Microsoft and Sun) can be seen as a confirmation.

Our work is based on and inspired by PeerWare [3], a middleware model developed within Saladin, though there are some differences w.r.t. the middleware model considered here.

A peer-to-peer middleware In the peer-to-peer paradigm the network underlying a distributed system consists of hosts at the same hierarchical level (*peers*).

A basic assumption in the middleware we consider is that mobility and distribution are encapsulated by the middleware, which provides to its users an abstract view of the current state of the network where all the details about the actual location of the resources and about the access modalities have been hidden. Thus, the user perceives the network as a collection of (passive or active) entities and the network changes as modifications of this community. Technically both PeerWare and our variation are in the stream of well-known coordination models based on shared data, like Linda, that have been recently adopted by middlewares for the distributed case, both in the client-server [6, 13] and in the peer-to-peer [3, 7] paradigm. The nature of the shared entities ranges from undetailed *tuples* in Lime [7] to *documents*, classified by *nodes*, in PeerWare [3]. Since we want to integrate this approach to handle mobility and distribution within an object-oriented paradigm, it is most natural to consider, as entities to be *shared*, standard *objects*. The object community of the overall system is distributed among the peers, so that we have several *local object communities*. The peers may be members of *groups*, that are communities of peers with the capability of reaching each other. Objects may be shared among the members of a group and their visibility is, hence, limited to the connected peers of that group. This provides means for dealing with privacy (only members of the group may access a visible object of that group) and for limiting the search space (a query for objects of some group does not involve the objects of the other groups).

Besides the shared objects, a distributed system consists also of communities of active and passive objects local to each peer that for privacy and security reasons should not be visible to the other peers.

UML-P2P: a middleware-oriented profile To help the designer of a distributed system based on the proposed coordination model, we provide a notational support, in the form of a UML profile, that we call UML-P2P, where the middleware will play the role of be the unique interface of each peer toward the external world and will take care of the visible object management.

Technically, in UML-P2P, we represent the middleware as an object of a predefined class, offering as operations the middleware primitives (see Sect. 1). More precisely, the middleware will be represented as a UML “interface”, namely a class with only operations and without instances. Besides that interface, a UML-P2P model (see Sect. 2) will consist of three structural diagrams describing the architecture of the overall system at increasing level of detail.

- The *Peer Diagram* shows the types of the peers used in the system and the cooperations needed among them to share the resources.
- The *Architecture Diagram* gives the peers and their organization in groups. Each cooperation between peer types stated in the Peer Diagram has to be matched in the Architecture Diagram by a common group between instances of those types.
- The *Network Deployment Diagram* describes the actual physical network, with the deployment of the instances of peers and their (possibly changing) connections. Of course, for the cooperation among members of each group to take place, appropriate physical connections have to be available. Thus, we will have to (statically) check the requirements from the Architecture Diagram against the network architecture as stated by the Network Deployment Diagram.

Then, for each type of peer present in the Peer Diagram the system designer must provide one *Peer Model* describing the part of the system resident on the peer of that type.

Semantic issues The “static semantics” requirements imposed by the profile on a UML-P2P model, that are partly informally discussed in Sect. 2, may be expressed as OCL constraints on the corresponding metamodel and hence are directly automatically checkable, e.g., by using a tool as USE [11], providing a valuable methodological help.

As for the semantics, we provide the definition of the middleware class in terms of UML. Thus the overall semantics relies on the semantics of (a restricted subset of) UML. It is well-known that this is a somewhat open issue; what we have done here is consistent with the standard way of proposing profiles in the UML community. We could have taken a more direct approach, giving the semantics for the proposed models following a formal approach, like the one outlined in [10].

To illustrate in a concrete way the use of the proposed profile UML-P2P, we use some parts of a running example.

1 The Middleware

1.1 Middleware in an OO World

The first key idea of our approach is that we represent the middleware as an object of a class predefined by the profile, offering as operations the middleware

primitives. In this way the access to the middleware primitives is uniformly disciplined by the standard mechanism of operation call.

It is quite common in practice the case of middlewares built out from different components in a hierarchy of types, each encapsulating some part of the system and offering an access to it at a higher-level of abstraction. This is, indeed, our case where the middleware consists of two layers: the first handling the connection and the disconnection of peers to groups and the other offering the primitives for accessing the local and the virtual global object communities of any group.

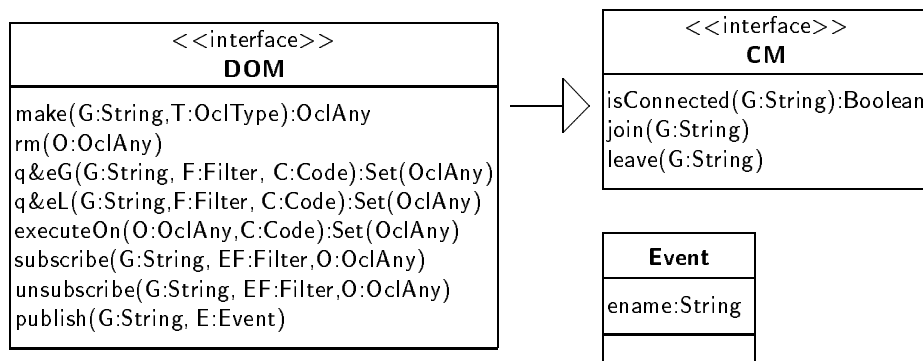


Fig. 1. The Middleware Interfaces

In a more general setting, for instance building profiles for software development based on a commercial middleware, like Corba, Jini or DCom, we will have a much larger class hierarchy, where each class will represent a simplified interface to the middleware offering only a part of the available services. Then the profile user will be able to select the minimal interface required by the application under development and will not have to understand (to pay for) and take into account the middleware features not needed in that specific case.

Technically, any variant is introduced by defining a UML interface, say *I*, whose operations are the primitives offered by such variant. Moreover, UML-P2P complements each of these interfaces with a UML class realizing it, say *I-Class*, with the constraint that in each peer there is always exactly one instance of *I-Class*. The UML description of the operations of *I-Class* (pre-post conditions, activity diagrams, ...), which is part of the profile definition, gives an abstract description of the middleware primitives; whereas the attributes of *I-Class* will give an abstract view of the information on the network managed by the middleware and on the current activities of the middleware itself.

In Fig. 1¹ we present the interfaces **CM** and **DOM** of the two levels, while the corresponding classes can be found in Sect. 1.2 and 1.3.

We want to stress that such classes are not intended to be actually implemented, but are only a precise description of the middleware interfaces. Therefore, we do not have any efficiency requirement to satisfy here, as this is not part of a modeling process. In particular, there are attributes whose values correspond to distributed computations and that would be, hence, too demanding on performances, if we had to actually implement them.

1.2 Connection Manager (CM)

The connection manager provides primitives for connection and disconnection to/from groups. Groups are logical communities of cooperating peers that have to be realized on some physical network. We consider two network paradigms:

- wired networks, that are characterized by the fact that, disregarding failures, each host is connected to the network till it explicitly disconnects.
- wireless networks, e.g., implemented via radio or infrared connections; changes in the physical distribution of the machines in the space may cause the connectivity to change without need for a conscious action from the involved peers.

Correspondingly, we distinguish groups in *wired* and *wireless* and hence the parameter of a (dis)connection implicitly determines the kind of connection. The primitives of **CM** are the obvious ones:

join connects to the given group.

leave disconnects from the given group.

isConnected checks if there is an active connection to the given group.

The CM-Class In Fig. 2, we give the **CM-Class** extending the **CM** interface from Fig. 1, where attributes and operations have been added to describe the semantics of the operations of the interface. Note that all the added features are *protected* (keyword **#**), that is they are accessible only from the class and its specializations. This will be true also at the next level. We will use objects of the auxiliary class **GroupData** to store the information on the kind of a group. Thus, we also include such class in the picture.

For each group there is, obviously, a unique object of class **GroupData** corresponding to it.

context **GroupData** **inv:**

`GroupData.allInstances->forAll(GD|GD <> self implies GD.name <> self.name)`

so that we can get the kind of a group from its name.

¹ The UML types **OclType** and **OclAny** correspond respectively to the set of all types of a UML model and to the type of generic objects.

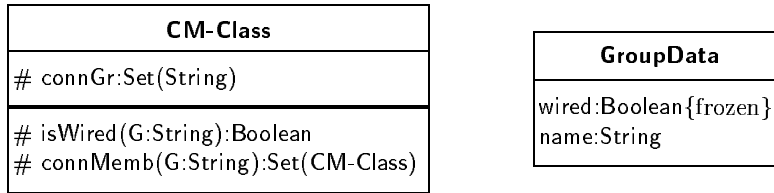


Fig. 2. The Connectivity Class

```

context CM-Class::isWired(G:String):Boolean
pre: GroupData.allInstances->exists(GD|GD.name = G)
post: result = GroupData.allInstances->select(GD|GD.name = G).wired

```

The attribute `connGr` contains all the groups to whom the `self` is connected.

```

context CM-Class::join(G:String)
post: connGr = connGr@pre->including(G)

```

```

context CM-Class::leave(G:String)
post: connGr = connGr@pre->excluding(G)

```

The operation `connMemb` yields for each group the set of peers currently connected; in particular the result is empty if the peer is not connected to that group. For the wired groups the result corresponds to all the peers currently connected to the group (that is, we assume wired groups to be deployed on a connected network), while for the wireless case the result corresponds to all the peers currently connected to the group within the range of the wireless connecting device. The latter condition cannot be expressed in OCL, at this level of abstraction, as we do not want to deal directly with data like distribution of hosts in the space or power of the connecting signal. So we use natural language to describe the result of `connMemb(G)` in the case when `isWired(G) = false`.

```

context CM-Class::connMemb(G:String):Set(CM-Class)
post: if connGr->includes(G)
  then
    if isWired(G)
      then result =
        CM-Class.allInstances->select(P|P.connGr->includes(G))->excluding(self)
      else "result is the set of all the peers P in the connecting range of self s.t.
        P.connGr->includes(G)"
      endif
    else result ->isEmpty
  endif

```

We can state a few properties that we know to hold for `connMemb` in the case of wireless groups: symmetry and the fact that each peer is visible on some group only if it is connected to it. The same properties hold also for the wired case, and, indeed, can be deduced from the definition of the operation.

context CM-Class **inv**:

```
connGr -> forAll(G | connMemb(G) -> forAll(P |
  P.connGr -> includes(G) and P.connMemb(G) -> includes(self)))
```

Note that

```
P.connMemb(G) -> includes(P')
```

implies that

```
P.connMemb(G) = P'.connMemb(G)
```

only if

```
isWired(G)
```

holds. Indeed, in that case both `P.connMemb(G)` and `P'.connMemb(G)` are the set of all the peers currently connected to `G`. But, if `isWired(G) = false`, then the physical distribution of the peers may be such that for instance `P'` is in the range of both `P` and some other peer `P''` connected to the `G`, but `P''` is not in the range of `P`. Therefore, in general, two peers connected to the same wireless group may have different views of the group state.

The operation `isConnected` is true if and only if the peer is connected to the group.

context CM-Class::`isConnected(G:String):Boolean`

post: `result = connGr -> includes(G)`

1.3 Distributed Object Middleware (DOM)

The DOM interface (see Fig. 1) is a specialization of CM, where the operations to deal with the shared distributed object communities have been added. The key point of our model is that each local object community is partitioned into a *private* and, for each group `G`, a (possibly empty) local *G-visible* part (shortly *LVOC(G)*). Then, the *G-visible* object communities local to `P` and to all the other peers connected to `P` on some group `G` are ideally merged together in a *G-virtual global object community* (shortly *VGOC(G)*), representing the shared data reachable by `P` as member of `G` at a given moment. A visual representation of the partition of the object community of a sample system is presented in Fig. 3. The system consists of 4 peers organized in 3 groups. The *VGOC(C)* is different for peer 2 and peers 3 and 4, because the former is at the moment not connected with such group.

The middleware has the absolute control of the visible object communities; in particular it creates and destroys such objects through the primitives:

make adds a new object of the given type to the local community of visible objects of the given group.

rm removes a given object from the local communities of visible objects.

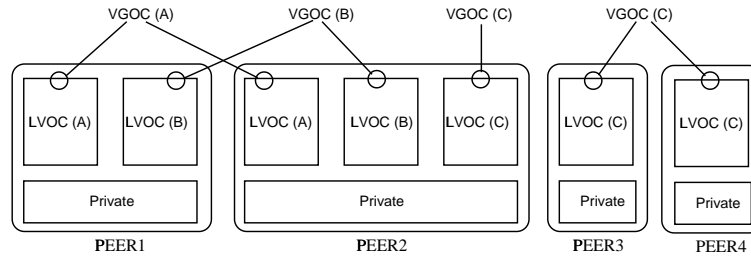


Fig. 3. Object Communities in a System

Visible objects can be accessed only through the middleware primitives, which are variations on the notion of filtering a collection of visible objects to get a set of objects of interest, apply some code to each of them and giving back the resulting objects to the caller. Thus, they are based on the auxiliary classes of `Filter` and `Code` that are both specializations of the (meta)class of the methods.

The elements of `Code` are the methods without parameters². s.t., in their body only public attributes, operations and methods of the owner class may appear. `Filter` is a specialization of `Code` corresponding to those methods where the result type is `Boolean` and that cannot have side-effects³.

Note that since `Code` and `Filter` elements must be statically correct in the context of their owner class (that, in any use of a `Code` in a search, will be the accessed visible object), in particular no references to the caller environment may appear in them. This on one side allows (efficient) remote evaluation, and on the other bans interlinks between private and visible communities (of different groups) and among the communities local to different peers, as the users cannot exploit (private) local object identities when assigning values to the attributes of (possibly remote) visible objects through code execution.

The primitives to access visible objects are the following:

q&eG (for query&execute Globally) given a group g , a filter f and a code c first selects the objects o in $VGOC(g)$ for which the call $o.f$ yields true. Then, it collects the results of $o.c$ for all such objects o , producing an object collection oc . It yields as result a deep copy of all the objects in oc , in order to prevent remote referencing.

q&eL (for query&execute Locally) given a group g , a filter f and a code c first selects the objects o in $LVOC(g)$ for which the call $o.f$ yields true. Then, it collects the results of $o.c$ for all such objects o , producing an object collection oc , and yields oc (without making copies).

executeOn given a local visible object o and a code c yields $o.c$ (without making copies).

² Technically, it is a specialization of the `Method` metaclass of the UML metamodel, where the unique parameter has kind `out`.

³ Precisely, they are UML queries.

The middleware provides also tools to manage a publish and subscribe event mechanism. The local private objects may subscribe to some event with their middleware instance, that will provide to propagate the subscription to the other instances. If an event is raised, then the middleware instance will notify it to all subscribers (both in the same and in other peers).

In the UML-P2P setting an event is just a special data, characterized by a name (a string). Technically, the events are defined by means of the stereotype <<event>> that is any class specializing `Event`, given in Fig. 1. The subscriptions to events are actually described in terms of *event filters*, that are filters whose owner class is an <<event>>, and correspond to subscriptions to all the events satisfying the filter. Let us briefly describe the primitives of **DOM** for the event mechanism:

subscribe given a group `g`, an event filter `ef` and an object `o` registers the interest of `o` to be notified whenever an event satisfying `ef` raises in the group `g`. The effect of the notification of `e` is to produce the call `o.handle(e)`.

unsubscribe given a group `g`, an event filter `ef` and an object `o` nullifies the effect of the call `subscribe(g,ef,o)`, if any, from that moment on.

publish given an event `e` and a group `g` raises `e` in `g`.

The DOM-Class Class It is a specialization of **CM-Class** and **DOM**, where the (private) attributes and operations to deal with the shared distributed object community have been added (see Fig. 4).

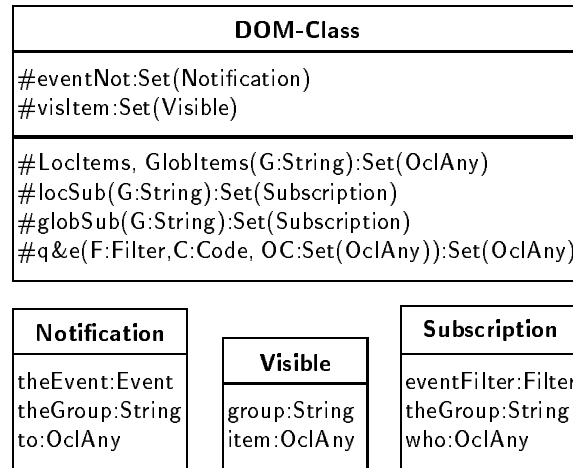


Fig. 4. The Distributed Object Class and its Auxiliary Classes

The Visible Object Communities The middleware is responsible for the management of the local visible objects, providing primitives to create, destroy and modify the objects in the local visible object community of each group. Each peer, that is each instance of **DOM-Class**, contributes a set of objects to the object community of each group, stored in the attribute `visItem`. Technically, the visible objects are objects enriched by the information of their owning group, as defined in class **Visible**.

The operation `make(G:String,T:OclType):OclAny` is defined by the following method.

```
{ o = create(T);
  v = create(Visible);
  v.item = o;
  v.group = G;
  visItem = visItem->including(v);
  return(o);}
```

The operation `rm` instead is qualified by the following constraints.

```
context DOM-Class::rm(O:OclAny)
pre: visItem.item->includes(O)
post: O.oclType.allInstances->excludes(O) and
       visItem.item = visItem@pre.item->excluding(O)
```

The operation `LocItems` yields for each group the set of objects corresponding to the visible objects of that group.

```
context DOM-Class::LocItems(G:String):Set(OclAny)
post: result = visItem->select(l||l.group = G).item
```

For each group, the union of the sets of visible objects local to all the peers currently connected is given by the operation `GlobItems`⁴.

```
context DOM-Class::GlobItems(G:String):Set(OclAny)
post: result = connMemb(G)->iterate(P:CM-Class;
  acc:Set(OclAny) = LocItems(G)| acc->union(P.LocItems(G)))
```

The above constraint is an idealized description and we can expect it to hold in practice only in those cases where the state of the group (both the connections

⁴ In the constraint for operation `GlobItems`, the OCL operation `iterate` is used. Its general syntax is

```
collection->iterate(elem :Type; acc:Type = <expression> |
  expression-with-elem-and-acc )
```

When the `iterate` is evaluated, first the accumulator `acc` gets an initial value `<expression>`; then `elem` iterates over the collection, the `expression-with-elem-and-acc` is evaluated for each `elem` and after each evaluation of `expression-with-elem-and-acc`, its value is assigned to `acc`.

and the local states of the individual peers) is sufficiently stable for the computation time to be irrelevant w.r.t. the changes. Indeed, the value of `GlobItems` corresponds to (and abstract from) a distributed computation: a visit of the connected peers in any order collecting their local visible objects. The constraints describe an ideal *atomic* computation at zero time, a view of the visible objects at one instant of the network life, while the result in practice could be the union of the views of the local visible objects of all peers taken in different points in time. The possible discrepancy between the value of `GlobItems` and the actual global state of the network will be reflected at the user level by the results of the global query operation, that is expressed in terms of `GlobItems`, but will be computed in all realistic implementation as a serialization of local queries on all the connected peers. However, we argue that in most cases this abstraction is *close enough* to the reality for the standard designer to be able to work on it. We also plan to provide a further extension of the middleware hierarchy with a “more realistic” `DOM-Class` class where the global queries are described directly in terms of the local ones.

context `DOM-Class::executeOn(O:OclAny,C:Code):Set(OclAny)`
pre: `visItem.item->includes(O)`
post: `result = O.C`

Searching Object Communities Let us first see the `q&e` primitive, that is private and instrumental to the description of both local and global query. A call `q&e(f,c,oc)` is statically correct iff the owner of `c` has a type conforming to the type of the owner of `f`, i.e., iff `c.owner.oclsKindOf(f.owner.oclType)` holds.

context `DOM-Class::q&e(F:Filter,C:Code,OC:Set(OclAny)):Set(OclAny)`
post: `result =`
`OC->select(oclsKindOf(F.owner.oclType))->select(O|O.F()).C()`

Using `q&e`, we can describe the searches operations offered by the interface.

context `DOM-Class::q&eG(G:String, F:Filter, C:Code):Set(OclAny)`
post: `result = q&e(F,C,GlobItems(G)).deepCopy()`

where we assume each user defined class to have an operation, `deepCopy`, creating a copy of the recipient and, recursively, of its object attributes, if any.

context `DOM-Class::q&eL(G:String, F:Filter,C:Code):Set(OclAny)`
post: `result = q&e(F,C,LocItems(G))`

The Event Mechanism We will use the auxiliary class `Subscription`, presented in Fig. 4. The elements of `Subscription` must satisfy the static requirement that the owner type of `eventFilter` is an `<<event>>`.

The sets of currently active local and global subscriptions are stored in the protected attributes `locSub` and `globSub`. If a peer `P` has an active local subscription for an event, then any other connected peer must have a subscription for the same event and `P`, and vice versa, as stated by the following constraint.

context P:DOM-Class **inv**:

```
P.locSub->forAll(ls|P.connMemb(ls.theGroup)->forAll(P'|P'.globSub->
exists(gs|gs.eventFilter = ls.eventFilter and gs.theGroup = ls.theGroup and gs.who = P)))
and
P.globSub->forAll(gs|P.connMemb(gs.theGroup)->includes(gs.who) implies
gs.who.locSub->exists(ls|gs.eventFilter = ls.eventFilter and gs.theGroup = ls.theGroup))
```

This condition abstractly describes the subscription propagation among the connected peers without imposing a particular reconciliation policy.

The local effect of a subscription is adding a new local subscription to `locSub`.

context DOM-Class::subscribe(G:String, EF:Filter, O:OclAny)

post: `locSub->select(ls|ls.eventFilter = EF and ls.theGroup = G).who->includes(O)`

The effect of an `unsubscribe` call is to remove the caller from the list of the local subscribers.

context DOM-Class::unsubscribe(G:String, EF:Filter, O:OclAny)

post: `locSub = locSub@pre-
(locSub@pre->select(ls|ls.theGroup = G and ls.eventFilter = EF and ls.who = O))`

Due to the class invariant requiring a synchronization among the lists of local and global subscriptions of the connected peers, the above constraint has also the effect of removing the peer where the unsubcriber is resident from the list of the global subscriber of all currently connected peers (if this is the only subscription for that event filter) and of all the other peers as soon as they get connected.

Analogously to the subscription, the middleware is also storing a list of notification messages, `eventNot`, with events that have been raised and subscribers (both local objects and other middleware instances) not yet delivered, using the auxiliary class `Notification`, presented in Fig. 4. The effect of publishing an event `e` is changing the state of `eventNot`, by adding a `Notification`, `n`, having `e` as `theEvent` for each subscriber. To get the set of the subscribers, we have to find the set of subscriptions for the event filters that are satisfied by `e`.

context DOM-Class::publish(G:String, E:Event)

post: `eventNot->select(n|n.theEvent = E and n.theGroup = G).to->includesAll
(locSub->select(ls|E.(ls.eventFilter)() and ls.theGroup = G).who->union
(globSub->select(gs|E.(gs.eventFilter)() and gs.theGroup = G).who`

Thus, each event raising creates a set of elements in `eventNot`. Then, we have to state that each of them is eventually dispatched to its addressee, corresponding to a call of the predeclared operation `handle`, and removed from the list or, if the addressee is a local object that's not anymore interested in such events just dropped from the list. Unfortunately this condition requires temporal logic and cannot, hence, be stated as an OCL constraint. At a greater level of detail, we could describe this requirement through the statemachine of an active object, part of the middleware, but here we prefer to state its abstract properties.

for each `n` in `eventNot`

global dispatch

```

If connMemb(n.theGroup) -> includes(n.to)
then eventually
n.to.eventNot -> includes(n) and
not eventNot -> includes(n)

```

local dispatch

```

If locSub -> select(ls|n.theEvent.(ls.eventFilter)()) and
n.theGroup = ls.theGroup.who -> includes(n.to)
then eventually
“the call n.to.handle(n.theEvent) will be issued”
and not eventNot -> includes(n)

```

unwanted notifications

```

If LocItems-(locSub -> select(ls|n.theEvent.(ls.eventFilter)()) and
n.theGroup = ls.theGroup.who) -> includes(n.to)
then eventually
not eventNot -> includes(n)

```

2 UML-P2P: a Middleware-Oriented Profile

So far we have discussed the nature of the middleware on which the application under development will be based and its description in our profile. In this section, we present the structure of a UML-P2P model. As stated in the introduction, such a model consists of one Peer Diagram, one Architecture Diagram, one Network Deployment Diagram and, for each peer type present in the Peer Diagram, one Peer Model.

In the following sections we will use as running example to present the use of UML-P2P the design of a distributed mobile application for handling the orders of a manufacture company: DORDERS. Such company stores the products in a few different warehouses, and handles the orders in some different branches. To send the invoices to the clients the company uses special mail centers that generate paper mail starting from electronic data. So far, the company network is wired, so that all the nodes are always available. But, the orders are collected by travelling salesmen, each of them equipped with a portable computer that may be connected to the company network by means of a modem. The salesmen are expected to occasionally connect to the network to transmit the orders, to download the updated catalogs, and to verify the availability of the products in some warehouses and the state of old orders. Moreover, salesmen may exchange documents about their trade union, when they meet, by setting up an ad hoc network by using the infrared port of their computers.

2.1 Peer Diagram

Usually peer-to-peer distributed systems consist of several peers (hosts) cooperating and sharing the common data, with different capabilities, that is, peers are *typed*. Typing the peers by means of a user-defined class-hierarchy, hence, allows the user to focus on each kind of peers at a time.

The Peer Diagram presents the types of peers used in the system and the needed cooperations among peers of the various types. The intuition behind the notion of cooperation is that each peer type needs (may need) some resources, made available by some other peer type sharing some group membership. We make explicit this dependency by the cooperation relation.

We will use two stereotypes:

`<<peer>>` (stereotype of Class) A `<<peer>>` is a class without attributes and operations.

`<<coop>>` (stereotype of Association) A `<<coop>>` is an association whose ends must both be of the stereotype `<<peer>>`, that must be navigable in both directions and that has no multiplicity constraints. It is visually depicted by a thick line.

In a Peer Diagram only classes that are of the stereotype `<<peer>>`, and associations that are of stereotype `<<coop>>` may be used.

We present in Fig. 5 the Peer Diagram of our running example. The peers of type **Branch** represent branches, those of type **WareHouse** the locations where the products are kept, those of type **MailCenter** the locations where paper mail is generated and sent, and finally those of type **Seller** the travelling salesmen. The cooperation connection between **Seller** and **Branch**, for instance, captures our intuition that salesmen need the branches to get the product catalog and, vice versa, the branches collect the orders to be processed from the salesmen.

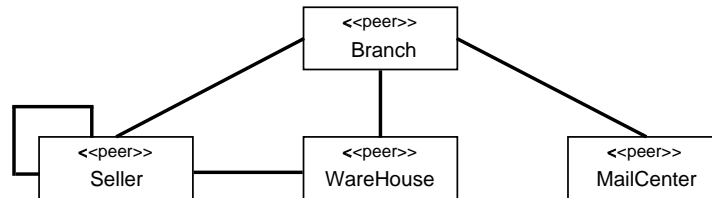


Fig. 5. DORDERS: Peer Diagram

2.2 Peer Model

A Peer Model describes the software required by the designed system on the peers of a given type. Hence, in a UML-P2P model there will be a Peer Model for each peer type present in the Peer Diagram.

Form of a Peer Model A Peer Model for a type of peers belonging to groups G_1, \dots, G_k consists of the following parts, see Fig. 6:

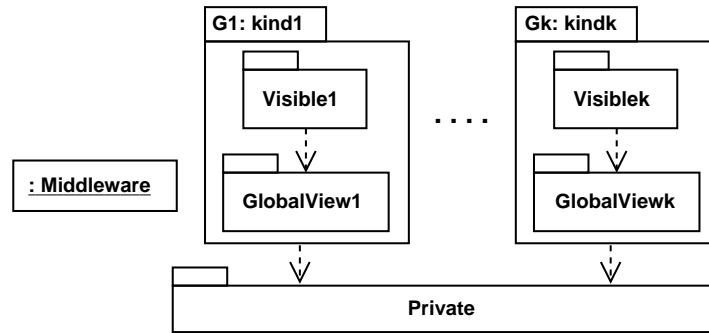


Fig. 6. Structure of a Peer Model

- a denotation of the used middleware, that is a *ClassifierRole* for one of the interfaces defined in Sect. 1. This choice, fixing the used middleware version, affects the static correctness of the calls to the middleware primitives in the other parts of the model.
- for each group in $G1, \dots, Gk$, a package named as the group itself with the qualification of the group kind (which may be either *wd* for wired or *wless* for wireless).
Each such package, corresponding to a group G , contains:
 - a package *Visible* that is a self-consistent class diagram describing the local visible object community $LVOC(G)$.
 - a package *GlobalView* that is a class diagram describing a view of the virtual global object community $VGOC(G)$. Since the local visible object community is part of the virtual global object community, the package *Visible* is imported by *GlobalView* (visually depicted by a dashed arrow)⁵.
- a package *Private* that describes the part of the system resident on the peers of that type. As the (local and global) visible communities are there to be used in the private part, the packages corresponding to the various groups are imported by *Private*. The class diagram in *Private* may contain active classes and statecharts to model their behaviour.

Peer Model for Seller As an example we present in Fig. 7 the Peer Model for the peer type *Seller* corresponding to the portable computers of the travelling salesmen. We will also take advantage of the example to illustrate some syntactic sugar provided by our profile.

We have collected in Appendix A the models of some of the other peer types of our running example.

Here we use a simplified notation to present the visible and the global view packages of a group: we depict their contents in two compartments within the

⁵ Here for simplicity we drop the decoration `<<import>>` over it.

icon of the group, instead of enclosing them in the package icon, with the visible package above and the global view below.

Moreover, we will use the following simplified notation to write filters, codes and events. A filter F is textually written `filter on T: cond` where T is F .owner and `cond` is F .body. Analogously a code C is textually written `do A on T returning RT` where A is C .body, T is C .owner, and RT is the return type of C . Whenever T is equal to RT , the latter is omitted, and so the code is simply written `do A on T`. Finally, an event E is textually written `E.ename(E.A1,...,E.An)` where $A1, \dots, A_n$ are the attributes different from `ename` of the class of E .

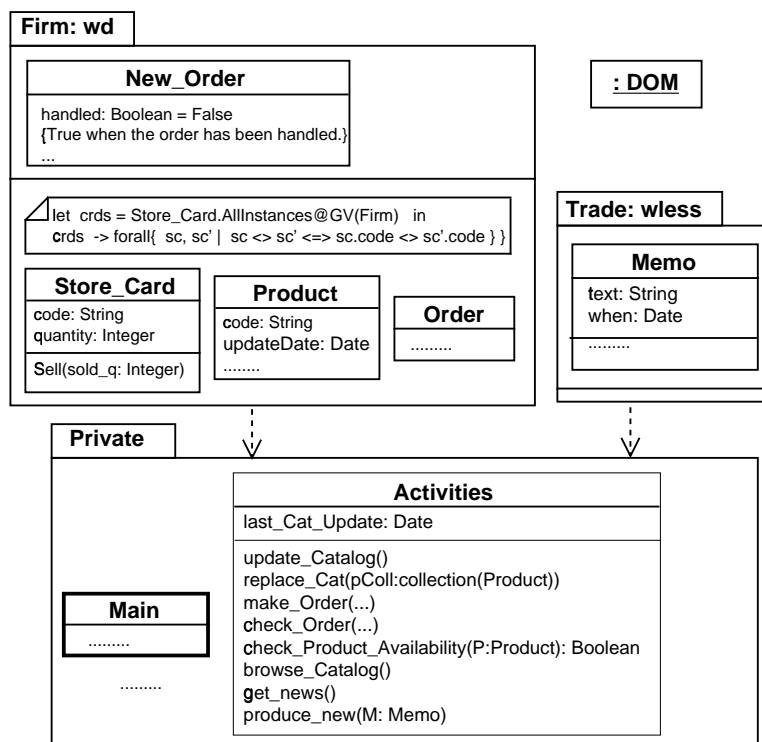


Fig. 7. Seller Peer Model

The sellers are members of two groups: **Firm** (wired) for cooperating with the other parts of the company, and **Trade** (wireless) used for cooperation with members of the trade union.

An object of the active class **Main** takes care to interact with the salesman through a visual interface and to call the operations of **Activities** corresponding to the most relevant activities, which are defined and commented below.

“`update_Catalog`” is defined by the method described below, which uses the primitives of the middleware, from both the connectivity layer (as `join`) and the higher-level (as `q&eG`) to retrieve the up-to-date information on the products from the network. In this paper we simply define a method by giving its body, as a UML note, by using UML actions.

```
{ if not DOM.isConnected(Firm) then DOM.join(Firm);
  pColl = DOM.q&eG(Firm,
    filter on Product: self.updateDate > [[self.last_Cat_Update]],
    do return(self) on Product);
  if pColl -> notEmpty then replace_Cat(pColl);
  self.last_Cat_Update = today; }
```

Notice that in the filter above the expression `self.last_Cat_Update` is enclosed by double square brackets. The meaning of such notation is that the enclosed expression will be evaluated by the caller, that is an object of class `Activities`, instead than being remotely evaluated on some object of class `Product`, as it is the case for `self.updateDate`. This notational facility is provided by the UML-P2P profile to allow in the body of a filter some subexpressions that have to be evaluated in the caller environment before moving the code (or the filter) to the possibly remote object for execution (filtering). The only restrictions are that such subexpressions are enclosed by `[[double square brackets]]` and that their type must be a datatype. Technically, this just corresponds to replacing in each call of the middleware primitives the local subexpression with another expression with the same value, say `V`, but without any reference to the caller state. If the type of the expression is a predefined type this is trivial; instead if it is a user defined datatype, then it is sufficient to create a copy of `V`. Obviously, the same notation will be used for the codes and the events as well.

The auxiliary operation `replace_Cat`, substituting the new for the old product information in the local mirror of the catalog, is instead abstractly modelled by the following postcondition.

```
context replace_Cat::pColl:Collection(Product))
post: Product.allInstances@P = pColl -> union(
  Product.allInstances@P@pre -> select(p | pColl.code -> excludes(p.code)))
```

Notice that above we needed to qualify the OCL operation `allInstances` to refer to those in the local private object community. Indeed, since the object community of a distributed system is partitioned by our coordination model into a plethora of communities, all the UML (OCL) constructs referring to “the object community” could intuitively become ambiguous in this context. Consider, as above, the construct `allInstances`. It could be interpreted as “all the instances in the distributed system”, or “all the visible instances of some group”, or “all the private instances”. To avoid any possible ambiguity, we introduce postfix markers (analogously to `@pre` in OCL postconditions) to qualify on which community we are operating. Thus, we will have that `allInstances@U` are all the instances in the universe (that is in the distributed system disregarding their physical

location), `allInstances@GV(G)` are all the (currently) visible instances in group `G`, `allInstances@LV(G)` are all the local visible instances of group `G`, and finally `allInstances@P` are all the local private instances. In UML-P2P model we require each occurrence of a construct referring to the object community to be qualified by one of the above. Of course, not all the qualifications do have sense.

The `GlobalView` package relative to the group `Firm` says that the information on the products (i.e., the catalog) are made available by other peers. In general, catalogs are provided by peers of type `Branch`, but, following the peer-to-peer philosophy, the actual location of an external resource is irrelevant. Indeed, our notation does not require to put this info in the model of `Seller`. The consistency verification of the overall model will successfully check that the peers of type `Branch`, belonging to the group `Firm`, make available objects of class `Product`, whose definition matches the one present in the global view package of `Firm`.

The operation `make_Order(...)` is defined by the following method

```
{ NO = make(Firm,new_Order);
  executeOn(NO,do self.handled = false; ... on new_Order)}
```

and corresponds to add a new element to `LVOC(Firm)`, describing the order collected by the salesman (the values of its attributes are set by using the primitive `executeOn`, and it would be incorrect, though harmless, to use the direct assignment `NO.handled = false`); such object is available to all the members of the group `Firm`.

The sellers belong also to the wireless group `Trade`, that they will use to exchange information about the trade union activity, when they meet, by setting up an *ad hoc* network, using the infrared port of their computers.

The operation `get_News` is realized by the following method, and just corresponds to copy all the available memos on the salesman portable computer.

```
{ if not DOM.isConnected(Trade) then DOM.join(Trade);
  ms = q&eG(Trade,
    filter on Memo: self.date > ... , do return(self) on Memo);
  ... save ms on the private community ...;
  DOM.leave(Trade)}
```

The operation `produce_New` simply corresponds to create a copy of a private object of class `Memo` in `LVOC(Trade)`.

The definition of the other operations of the class `Activities` can be found in Appendix A, together with the Peer Models of the other types.

Static correctness of a Peer Model in isolation Following the intuition that the objects in the visible communities are data manipulated by the system through the middleware, all the classes in the packages `Visible` and `GlobalView`, must be passive, and no calls of the middleware primitives are allowed in `GlobalView`, nor in `Visible` for some group `G`, but those for the `publish` primitive, with `G` as first parameter.

Analogously, since we have the intuition that the event mechanism provided by the middleware is to be used by the local activity only to monitor the visible and the global virtual object communities, the objects of the classes in **Private** may use all the operations of the picked middleware variant, except **publish**. They may subscribe to all events of classes listed in the package of some group and event classes may appear only in the **Visible** and in the **GlobalView**, to describe which are the events that may be published by the visible objects.

The calls to the middleware primitives have to satisfy some quite obvious static correctness as well. For instance **subscribe(g,ef,o)** is correct only if the owner of **ef** is of event type, **o** is the caller and the class of **o** has a **handle** operation with one parameter of type **Event**. Analogously, **unsubscribe(g,ef,o)** is correct only if the owner of **ef** is of event type and **o** is the caller. Since the last argument of any **subscribe** (**unsubscribe**) call made by some object will be the object identity (i.e., we do not allow an object to subscribe the interest of *another* object), such calls will be written in any UML-P2P model without the (un)subscriber argument and will be expanded in calls with the **self** as object argument.

Queries as well have to satisfy some consistency requirement among their parameters; that is, **q&eG(g,f,c)** (**q&eL(g,f,c)**) is statically correct only if the class of the owner of **f** is a specialization of the class of owner of **c** (so that the filtered objects are able to execute the code) and both classes are in the **GlobalView** (**Visible**) package of **g**. Moreover, **executeOn(o,c)** is statically correct only if the class of **o** is a specialization of the class of the owner of **c**.

Obviously, the calls of the middleware primitives present in the private part acting on the local visible or on the global community of some group must be correct w.r.t. their descriptions given in the respective packages. For example, it is possible to subscribe to some event in a group **G** only if the event class is present in the **G** package, and to leave and to join only the groups, for whom a package belongs to the model.

Notice that the designer has to explicitly mention in the **GlobalView** only those classes and constraints that (s)he is going to use. But, in most of the cases, the *VGOC* will be larger than it could be inferred from the **GlobalView**.

Static correctness of the Peer Model family The **Visible** packages of all the Peer Models, disregarding their group, must be pairwise consistent, that is, classes, associations and events with the same name appearing in two of them must be defined exactly in the same way in both. This condition allows us to define a “class diagram for the global virtual object community”, say **GVCD**. Hence, for each group **G**, the restriction of the **GVCD** to those elements present in at least one **Visible** package for **G** of some Peer Model yields a “class diagram for the global virtual object community of group **G**”, say **GVCD(G)**.

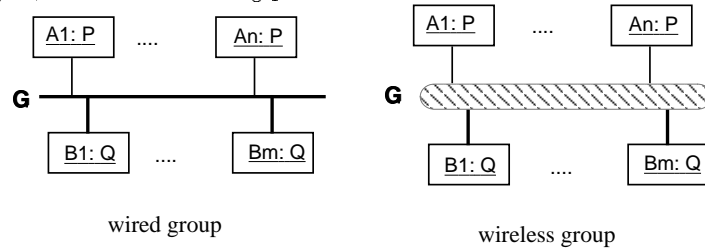
For each group **G**, the **GlobalView** of each Peer Model must be consistent with the **GVCD(G)** defined above, that is all classes, associations and event in the first must be present in the latter and defined exactly in the same way. Moreover, the matching class (association, event) in the **GVCD(G)** must be reachable from

a peer of such type by using the `<<coop>>` association. In this way we can check, at least at the peer type level, that the assumptions of any peer on the cooperative effort of the other peers are valid.

2.3 Architecture Diagram

The Architecture Diagram describes the structure of the designed system by saying which are the peers composing it and how they are organized in groups. The peers, which are instances of the peer classes presented in the Peer Diagram, are represented as **ClassifierRole**.

A group G is represented by a line to whom its members are connected. If G is wired, then the line is a black segment, else a shadowed rectangle with rounded angles, as in the following picture.



Whenever we know that a peer is permanently connected to a group G (i.e., it joins G immediately after being created and never leaves G), we make thick the line connecting it to the group. For example, in the above pictures, all peers of type Q will never leave the group G .

We further assume that all peers composing the designed system must appear in the Architecture Diagram. If their number is not determined a priori, we can attach to the object icons multiplicity indicators limiting the number of instances. Thus, we can express precisely the architecture of the system by means of this diagram.

If the Peer Diagram states a cooperation between two peer types, say $P1$ and $P2$, then for each instance of one of them there is an instance of the other in the Architecture Diagram s.t. they are members of a common group. This condition guarantees that the system architecture is able to support all the cooperations required in the Peer Diagram.

In Fig. 8, we present the Architecture Diagram for the DORDERS system. In this picture we use the multiplicity annotations to state that there are 98 mail centers and from 1 to 1000 sellers.

2.4 Network Deployment Diagram

The Network Deployment Diagram is a stereotype of “instance level collaboration diagram” describing which kind of network will be used to realize the designed system, and how the peers composing the system (defined in the Architecture Diagram) are deployed on such network. The objects correspond to

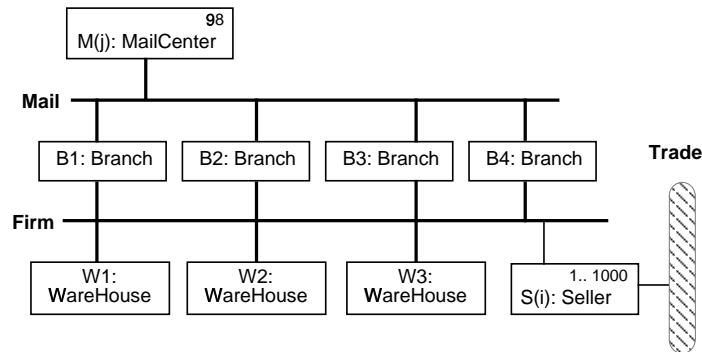


Fig. 8. DORDERS: Architecture Diagram

the peers and are instances of the peer types, presented in the Peer Diagram. A stimulus between two peers correspond to the existence of a network connection between them. We consider three kinds of connections:

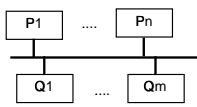
1. persistent during the life of the peers and unbreakable by them;
2. persistent between an explicit connection and disconnection decision of a peer;
3. possibly available, but not guaranteed, between an explicit connection and disconnection decision of a peer, e.g., a wireless network.

Clearly, here we do not consider the fact that a connection may be always broken due to a failure. Those three kinds correspond to three stereotypes of stimuli and are visually depicted by a thick line, a normal line and a dashed line respectively.

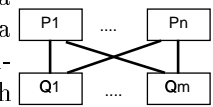
We further assume that all peers composing the designed system must appear in the Network Deployment Diagram. If their number is not determined a priori, we can attach to the object icons multiplicity indicators, as in the Architecture Diagram.

In Fig. 9, we present the Network Deployment Diagram for our running example. The underlying network consists of stable connections among all the branches and all the mail centers (e.g., by Internet). The branches are also variously connected by some corporate net to the warehouses. Instead, the sellers may dynamically connect to the branches, e.g., by a modem, and to other sellers by an infrared connection.

As a notational shortcut, we draw



to present in a compact way a completely connected graph as



For any group present in the Architecture Diagram, its members should be connected in the Network Deployment Diagram by a chain of communication links of the appropriate kinds. This condition guarantees that the deployment of

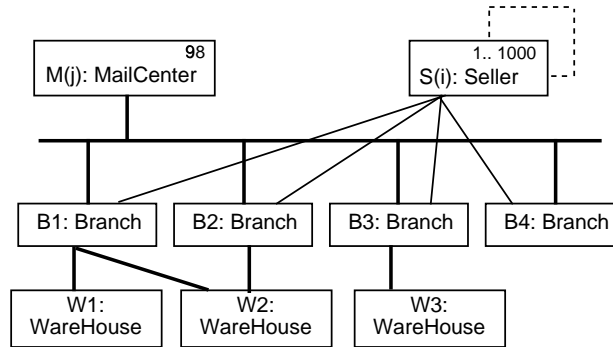


Fig. 9. Network Deployment Diagram

the system over some network is able to support all the groups activities required in the Architecture Diagram.

For instance, in our example the branch **B4** is connected to other members of the group *Firm*, those of class *Warehouse*, by a chain of links going through some other branch.

3 Conclusions and Future work

From the experience of some project dealing with significant case studies, we have been impressed by the gap existing between the proposed rigorous software development techniques and the use of the various kinds of middleware in the industrial practice. The gap has been noted also in the literature ([4]), where the use of middleware-oriented methods in software engineering has been advocated. With that goal in mind, we have presented a UML profile, UML-P2P, for a visual notation supporting the development of software for a distributed and mobile environment based on the use of a specific peer-to-peer middleware, inspired by PeerWare [3].

The idea of representing the middleware as an object is general and powerful enough to be reused whenever designing a UML profile to support a development process based on some middleware. Another benefit of this approach is that the description of the middleware class plays for the profile user the role of a reference manual, in the very language the user is familiar with.

We plan to investigate how to develop UML profiles for other middlewares following the ideas of this paper, that require to find out

- the primitives provided by the middleware;
- a precise but quite abstract description of their arguments;
- a precise but quite abstract description of the network as perceived by the middleware users.

For example, for the case of Jini, we should characterize and define using UML the “Jini services” and the requests for a service of some kind.

In the literature there are already proposals of UML profiles for supporting the use of a particular coordination model for distributed systems, see e.g., [1] for the case of tuple spaces. Other attempts, in the literature, at developing UML notations for supporting the use of a specific middleware concerns CORBA; currently an official proposal is under discussion at the OMG, see [9]. The main difference of [9] with our approach is that it gives a UML way to define the interfaces written using CORBA IDL, whereas the other aspects of the system should be modelled as usual when using UML. We think that if we apply our approach to CORBA, perhaps we would get a more abstract view of the “CORBA components” and a notation taking care of the design of the whole system; thus we would consider also some methodological aspects of its use.

In the future we plan also to extend the UML-P2P notation by attaching some information concerning the “quality of service”, as for instance the probability that a peer is reachable from another one, thus allowing us to make performance analysis of the designed system.

Acknowledgments We acknowledge the benefits of many discussions with the colleagues of the Saladin project, and especially the designers of PeerWare, G.P. Cugola and G.P. Picco.

References

1. E. Astesiano and G. Reggio. UML-SPACES: A UML Profile for Distributed Systems Coordinated Via Tuple Spaces. In *Proc. ISADS 2001*. IEEE Computer Society Press, 2001. Available at <ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio00a.pdf>.
2. J. Charles. Middleware Moves to the Forefront. *Computer*, 32(5), 1999.
3. G. Cugola and Gian P. Picco. PeerWare : Core Middleware Support for Peer-to-Peer and Mobile Systems. Manuscript, submitted for publication, 2001.
4. W. Emmerich. Software Engineering and Middleware: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.
5. G. Reggio, M. Cerioli, and E. Astesiano. UML-P2P: A Visual Notation for the Development of Peer-to-Peer Mobile Applications. Technical Report DISI-TR-01-54, DISI, Università di Genova, Italy, 2001. Available at <ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll101a.pdf>.
6. Sun Microsystems. JavaSpaces Specification. Technical report, Sun, 1999.
7. A. Murphy, G. Picco, and G-C. Roman. Developing Mobile Computing Applications with Lime. In M. Jazayeri and A. Wolf, editors, *Proceedings of the 22th International Conference on Software Engineering (ICSE 2000), Limerick (Ireland)*. ACM Press, 2000.
8. OMG. White paper on the Profile Mechanism – Version 1.0. <http://uml.shl.com/u2wg/default.htm>, 1999.
9. OMG. UML Profile for CORBA, Version 1.0. <http://www.omg.org/cgi-bin/doc?ad/00-02-02.pdf>, 2000.

10. G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001 - Fundamental Approaches to Software Engineering*, number 2029 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.
11. M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In S. Kent A. Evans and B. Selic, editors, *Proc. UML'2000*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.
12. UML Revision Task Force. *OMG UML Specification*, 1999. Available at <http://uml.shl.com>.
13. P. Wyckoff. TSpaces. *IBM Systems Journal*, 37(3), 1998.

A Example DORDERS

Peer Model for Seller(completion) The operation `check_Order` is realized by the following method, and just corresponds to search particular elements of the global virtual object community, and to show them on the salesman portable computer.

```
{ if not DOM.isConnected(Firm) then DOM.join(Firm);
  oColl = q&eG(
    filter on Order: self.date = ... and self.code = ... and ...
    do return(self) on Order);
  ... show by a visual interface oColl ...}
```

The operation `check_Product_Availability` is quite similar and is defined by:

```
{ if not DOM.isConnected(Firm) then DOM.join(Firm);
  intColl = q&eG(Firm,
    filter on StoreCard: self.code = [[P.code]],
    do return(self.quantity) on StoreCard returning Integer);
  return((intColl -> Sum) > 0);}
```

Peer Model for Branch The peers of type `Branch` correspond to the various branches of the company. Each branch takes care of different kinds of products, and must make available the current catalog of its products and to retrieve and process the orders corresponding to them. The orders are retrieved from the peers corresponding to the salesmen, when they are connected to the company network. Then, if the products are available, they are sent to the client, and by using the mail centers a paper invoice is generated and mailed to it.

Notice that these peers may handle events raised in the group `Firm`, and that they may raises events for the members of the group `Mail`. Clearly the operation `handle` needed to handle the events is implicitly defined for any class appearing in the peer model; moreover on the transitions of the statecharts we simply write the UML-P2P event `E`, and not the UML event `handle(E)`. The peer model for `Branch` is reported in Fig. 10. In this case, because the package `Visible` for the group `Mail` is empty, the corresponding slot is empty.

We design the behaviour of the elements of the active class `Order_Handler` by the statechart reported in Fig. 11, while we report and comment below the operations of the auxiliary static classes `Retriever` and `Invoicer`. Instead we do not further detail the class `Catalog_Handler`.

The operation `retrieve_Orders` is defined by the following method.

```
{ noColl = q&eG(Firm
  filter on New_Order: self.handled = false and
    [[self.own_Products]] -> includes(self.code),
  do self.handled = true; return(self) on new_Order);
  while (noColl -> notEmpty) do
```

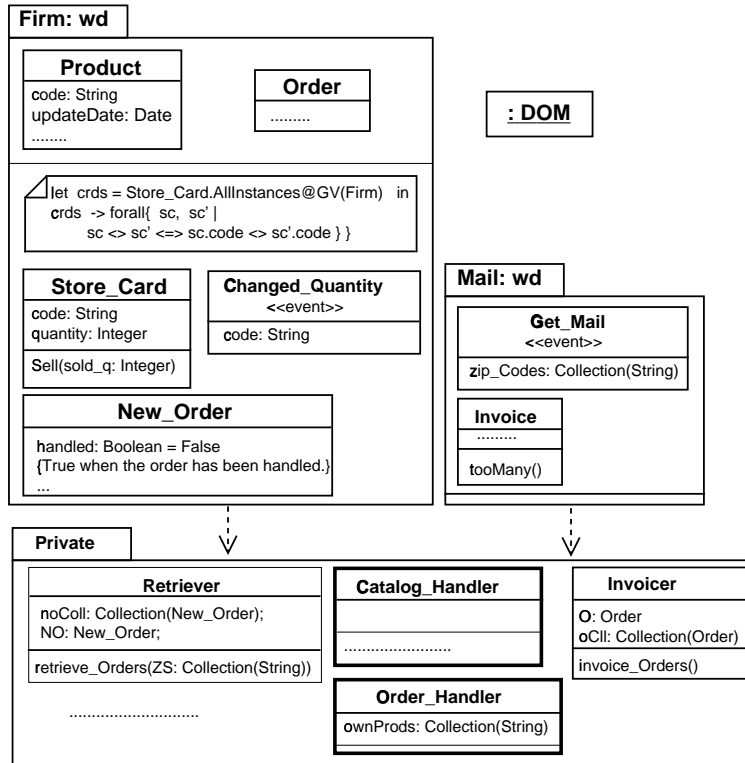


Fig. 10. Branch Peer Model

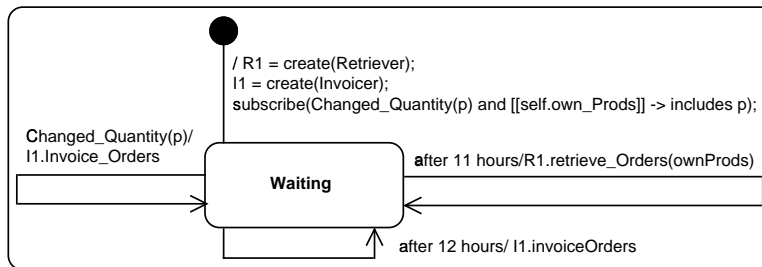


Fig. 11. Behaviour of Order_Handler

```

{NO = noColl.first;
 O = make(Order);
 executeOn(O,do self.code = [[NO.code]], self.status = Pending, ... on Order);
 noColl = noColl ->excludes(NO)}

```

and Invoice_Orders by

```

{ oColl = q&eL(
  filter on Order: self.status = Pending, do return(self) on Order);
 while (oColl ->notEmpty) do
  { O = oColl.first;
  sc = q&eG(
  filter on Store_Card: self.code = [[O.code]],
  do if self.quantity > [[O.quantity]] then self.Send([[O.quantity]]) on Store_Card
  l = make(Invoicer);
  executeOn(l,do self.... = ...; ...; self.tooMany on Invoicer)}}

```

The method associated to the operation `tooMany` of `Invoicer` is defined as follows, and takes care to raise the event `Get_Mail(zip)` whenever there are too many invoices waiting to be mailed:

```

{ let invs = Invoice.allInstances@GV(Mail) in
  if invs ->select{ l | l.handled = false } -> size > 1000 then
  publish(Get_Mail(invs.zip));}

```

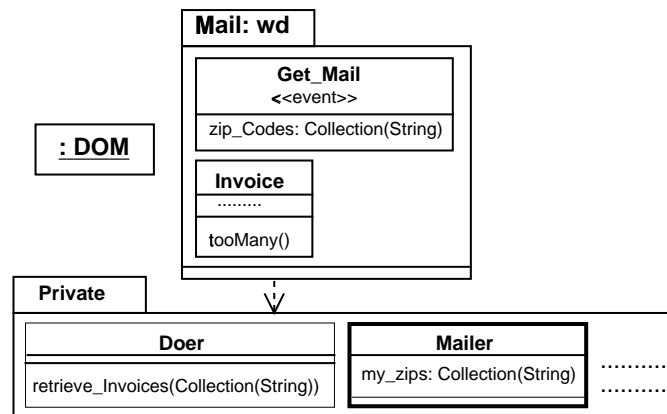


Fig. 12. MailCenterPeer Model

Peer Model for MailCenter A mail center is a plant of the Mail Service able to generate paper mail starting from electronic documents; we present the corresponding Peer Model in Fig. 12.

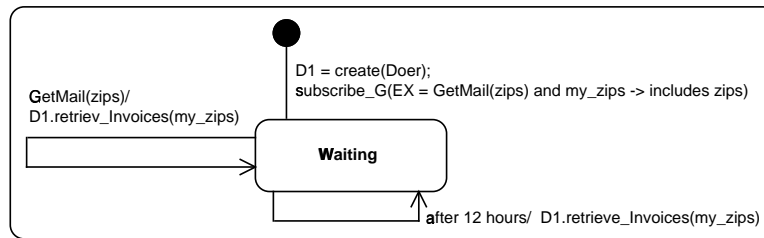


Fig. 13. Behaviour of Mailer

We design the behaviour of the elements of the active class **Mailer** by the statechart reported in Fig. 13. The method associated with the operation `retrieve_Invoices` of the auxiliary class **Doer** is defined below.

```

{ iColl = q&eG(
  filter on Invoice: self.handled = false and
  [[Mailer.my_zips]] -> meet self.zip) -> notEmpty,
  do self.handled = true; return(self) on Invoice
while (iColl -> notEmpty) do
  {l = noColl.first;
  ... generate paper invoice from l ...;
  iColl = iColl -> excludes(l)}}
  
```