An Architecture and a Process for Implementing Distributed Collaborations

Eric Cariou [1,2] Antoine Beugnard [1] Jean Marc Jézéquel [2]

[1] *ENST Bretagne, BP 832*
29285 BREST CEDEX, FRANCE
Antoine.Beugnard@enst-bretagne.fr

[2] *IRISA/Université de Rennes 1*
Campus de Beaulieu
35042 RENNES CEDEX, FRANCE
{cariou,jezequel}@irisa.fr

Abstract

Collaborations (between objects) are increasingly being recognized as fundamental building blocks to structure object-oriented design, and they have made their way into UML. But very often the first class aspect of a design level collaboration is lost during the detailed design process, making it difficult to keep good traceability between the design and the implementation. The problem is not simple, because for any given collaboration abstraction, there might be several possible design solutions depending on the many non-functional forces impacting a given application. We propose a process and an architecture in which the notion of collaboration is preserved from analysis to design and implementation, while allowing the designer to change his mind about which particular design trade-off is selected in order to face changing non-functional requirements during maintenance. We illustrate our approach with a case study inspired by the real example of a large French railway company attempting to adapt a flight reservation system to its own context.

1 Introduction

The interest of the notion of *collaboration* has long been recognized in the object-oriented community, and some methodologies of the early nineties (such as CRC [17] or OOram [12]) even concentrated on collaborations as the basic building blocks to carry out object-oriented design. Collaboration diagrams are now well established as one of the core components of UML [11]. They allow the easy handling and reusing of *interaction abstractions* among components, and parameterized collaborations serve to represent the application of design patterns [15].

But the first class aspect of a design level collaboration is generally lost during the detailed design process, making it difficult to keep good traceability between the design and the implementation. At implementation level, very few traces of these abstractions are still visible: collaborations have been refined, split and lost in a set of objects that can be distributed over a network and communicate through "low level" primitives such as remote procedure calls.

In the context of distributed systems, a major important issue is precisely the collaboration or interaction implementations between components. Unfortunately, systems and algorithms for communication among remote components can be highly dependent on non-functional constraints (from the point of view of the "interaction semantics"): the location and number of interacting components, the state of the network used, the need to encrypt communication or to have a fault-tolerant system, etc. For instance, in an event broadcast system, the fact that the data transmitted are encrypted or not, does not change the way the components use the system; they will still call the same services. But the implementations of these two versions of the same interaction abstraction are quite different.

Then, when there is a change in the deployment context of the application where non-functional constraints are involved and new design choices are selected for implementing the collaborations, it can be extremely difficult to disentangle the functional code from the collaborative one because the two are more or less mixed. In this paper we propose a process and a framework where the notion of collaboration is preserved from analysis to design and implementation, while allowing the designer to change his mind about which particular design trade-off is selected to face changing non-functional requirements. We illustrate our approach with a case study inspired by the real example of a large French railway company trying to adapt a flight reservation system to its own context.

The rest of the paper is organized as follows. In section 2, we discuss a seat reservation application in different contexts and show how the number of components involved can have a major influence on the implementation. In section 3, we propose an organization of components that

leads to an architecture where communication abstractions are explicit. We introduce interaction components and we explain in section 4 how they can be specified using UML collaborations. Section 5 describes implementation variants of these interaction components in order to illustrate how the management of non-functional constraints could be improved. We discuss related works in section 6, before concluding with some interesting perspectives for interaction components.

2 The story of a seat reservation application

Let us imagine a small company, ErnestCo, whose activity is to make seat reservations for a small bus travel company. ErnestCo has developed an application to manage the whole reservation process. This application is such a success that a flight company wants to acquire it in order to deploy it in its hundreds of agencies. Later, a famous railway company wanting to reorganize its reservation system decides to acquire this application and install it in its tens of thousands of ticket offices.

The main problem the reservation application has to tackle is scalability. How can an application designed to solve small size problems be deployed to solve very large (geographically and number of data) ones? The first version of the system can run on a single server, the second one needs to be widely distributed throughout the world, and the last one, although not as widely distributed as the second one, has to manage many more access points.

This case study was inspired by a real life situation: some years ago, the French national railway company acquired the reservation system of a Canadian airline company in order to use it as its new reservation system. The adaptation of the system was a very complex task because of the important change in context of use.

In all three applications the reservation interaction can be described as follows: components communicate through a shared memory containing identifiers (representing seats in a bus, a plane or a train). Each identifier is unique and a component can reserve an identifier and cancel this reservation. If an identifier is reserved by a component, it is not available for reservation by the other components. Once the cancellation of this identifier reservation is made, it is added to the set of identifiers available for reservation. Some components can be informed about the number of available identifiers in the system.

The key point of the design is the responsibility of the interaction. Where is the frontier of the interaction? The separation between the communication system and its clients is crucial. The proposed solution has the responsibility of managing identifiers inside the interaction. If identifiers were managed by an external component, the whole architecture of the system would have to be redefined for each problem size. Putting identifiers *into* the interaction system hides the way they are managed; they are in it, but implementation and design choices give their true localization and management process; they can be distributed or not, duplicated or not, load-balanced dynamically or not, etc.

In the following, we describe the use of this reservation interaction in two contexts and discuss the consequences of such a communication reification on non-functional features.

2.1 The bus seat reservation application

A bus is composed of a set of seats. Each seat is referenced by a unique identifier (e.g., a number). The set of these identifiers makes up the identifiers of the interaction described above. A traveler in a bus occupies one identified place.

Components interact among themselves in the following way through the reservation interaction:

- When a traveler wants a seat in a bus, the agency sends a query to obtain a seat identifier. If the bus is full, a special value is returned.
- When a traveler cancels a reservation he previously made, this seat is again available for reservation.
- When a bus has started its journey, the agency is informed that reservations for this bus are closed.
- When the company opens reservations for a bus, the agency is informed of this new journey.

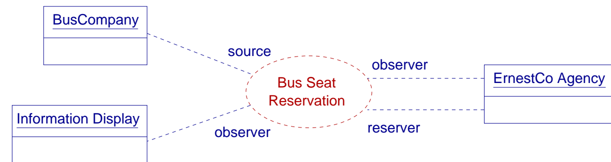

Figure 1. Bus seat reservation application

In UML, interactions between classes or objects can be specified by a collaboration. We find the use of the collaboration representing the identifier reservation interaction (called `Bus Seat Management`) in the UML instance diagram of figure 1. This figure describes the bus seat reservation management system. We can see in this figure the reservation interaction (represented by the `Bus Seat Management` collaboration), the ErnestCo agency that sells seat reservations for the clients of the bus company.

Hence, these two components interact through the `Bus Seat Management` collaboration. In this collaboration,

the ErnestCo agency plays a reserver role because it can reserve (and cancel) a seat in a bus and plays an observer role because it may ask for information on the number of available places in a bus. The bus company plays the source role because it sets and removes data (number and identification of seats) about buses; these data are managed by the collaboration. A display (playing an observer role) shows the state of the reservations for all the buses.

2.2 The flight seat reservation application

The previous interaction through an identifier reservation system is not specific to the bus seat management application. It can be *reused* in other contexts. For instance, if the data to reserve are flight seats instead of bus seats, the collaboration can be used in a context of a flight seat reservation system. An airline company that wants to sell places on its flights then plays the source role. Figure 2 shows this application. The reserver and observer roles are travel agencies that reserve seats on these flights for their clients and the source is the airline company ErnestAir.

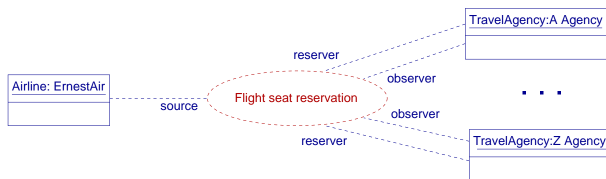

Figure 2. Flight seat reservation application

The context of this application is completely different but the nature of the interaction is the same as in the bus seat management application. The collaboration used will be a slightly-modified version of the former collaboration.

We can notice that in figures 1 and 2, representing the bus and the flight seat reservation applications, the seat identifier sets are not visible. Indeed, they are managed inside collaborations. An intuitive way to design these applications would have been to give the responsibility of managing the seat identifier sets to the bus company (in the case of the bus seat reservation application) as shown in figure 3.

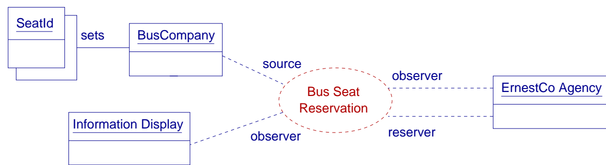

Figure 3. A bad bus seat reservation application design

But this design is rather bad. The complexity of the application is the data set management. As we shall see in the following sections, in order to face different contexts of use, different implementations of this data set management have to be defined. But here, we have already chosen an implementation design (all the data are managed by a single component) that will certainly not fit all situations. So putting the set management in the collaboration hides the way they are managed and makes the interaction more reusable and "interesting". Several designs of this collaboration can be made to face different contexts of use without changing the way of using the collaboration. A more complete discussion of this problem of interaction responsibility is available on our web site[1].

2.3 Discussion

These two applications are both similar and different. Functionally both applications have the same requirements, but their non-functional requirements are very different.

The major implementation problem is the management of the identifier sets. The way it is done is highly dependent on the context in which the interaction is used. Notably the distribution and the number of components have a major influence on the implementation. Indeed, if the application is deployed over a local network and is composed of only a small set of components, like for the bus seat reservation application, a centralized management on a single server is efficient enough and is simple to implement. However, if this application is running over the Internet or a wide area network and is built of thousands of components, like the flight seat reservation application, this simple interaction system cannot be used, because the single server would be a bottleneck. One solution is to distribute a subset of all identifiers to each reserver component. This will lead to using protocols allowing free identifiers to be sent to those components whose local set is empty.

Another difference lies in the dynamic feature of the system. The bus seat application is completely static. The components that form the application are well known. There is a single agency that sells seat reservations and that will never change. The flight seat reservation application is different since the components are widely distributed all over the world. Their number can be large and some new components (some travel agencies) can connect to (and disconnect from) the system after the beginning of the reservation process. The implementation has to deal with these dynamics that make the distribution of the sets more difficult because the number of components can change.

[1] `http://www-info.enst-bretagne.fr/medium/` section "Introduction to communication components from the point of view of UML designers"

So, although the interaction remains the same in principle, the way it is implemented is highly dependent on non-functional constraints and requirements: the number and localization of the components, the dynamics of their connection to the system, the capacity to resist a load increase, the necessity to encrypt the transmissions or the availability of the application for instance, lead to a broad spectrum of design and implementation solutions.

An important issue is, then, how to implement a specialized interaction. Usually, the interaction management is "embedded" in the components themselves. The implementation of the interactions is a part of the component. The code of the interaction and the code of functional concerns are more or less mixed. That leads to a bad separation between the functional and the interactional elements. Hence, it is difficult for the component to easily use another implementation of the same interaction if non-functional constraints change between two contexts of use.

In summary, the complex problems we have to face are the following:

- How can we implement an interaction "correctly"?
- How can we easily change an interaction implementation if the context of the application has changed?

The following sections present the solution we propose. It is based on the reification of interactions into components both at specification and implementation level allowing the separation of functional and interactional concerns throughout the software development process.

3 The interactional concern at the implementation level

In this section we study an interactive video application and we focus on the implementation of the component interactions. In this application, a server broadcasts movies, watched by some viewer components. Just after the end of each movie, the server asks the viewers to choose the next movie they want to see by launching a voting session. Then, once the vote is finished (i.e., when all the viewers have voted or when the vote session time has elapsed), the next movie is broadcast, according to the majority choice.

3.1 Architecture of the application

If there are three viewers or clients, four components interact among themselves in this application: the three client components and the single server component. For their communication, these four components use middleware such as Corba[14] for instance. It allows method or service calls on remote components. Other kinds of middleware could also be used. Some glue is needed to adapt the components according to the way the middleware is used. In the case of Corba, this glue is the stubs and skeletons that must be generated in order to be able to call a remote method.

Concerning the communication inside the application, it deals with two kinds of interactions: the broadcast of the video stream and the vote for the next movie. The realization of these two interactions implies communication between the remote components and executing tasks such as stream encoding and decoding in the case of the video broadcast. Thus, for each interaction, some "piece of code" is associated with the functional code of the components. The architecture of this application is shown in figure 4.

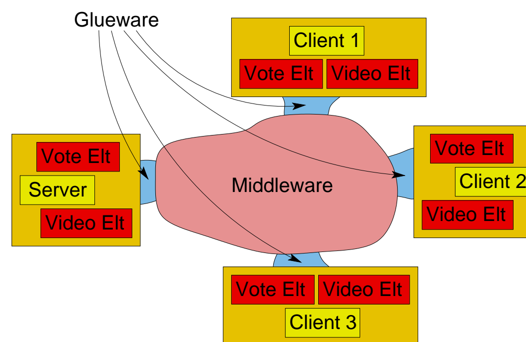

Figure 4. Detail of the interactive video application architecture

A component is composed of several parts such as the elements managing the interactions (video broadcast and vote in our case, the elements being named `Video Elt` and `Vote Elt` in the figure) and the "functional" part of the component.

3.2 Discussion of the architecture

This classical way of building components in a distributed context presents certain drawbacks. First, the vote and video broadcast interactions are not completely specific to this interactive video application. Indeed, we can easily imagine using the video broadcast system in a video-conferencing application as it also needs to broadcast video streams. But, if the software designers and developers have not build this interaction system in terms of an independent and reusable system, the result will be unsatisfactory. The code of the functional part and the code of the interactional parts will be more or less mixed. This leads to a bad decoupling of the functional and the interactional concerns at the implementation level, making the interaction system unreusable.

And since these parts are badly decoupled, if a change is made, for instance, in the vote interaction on the server side and also the according modification of the code of the vote interaction on the client side, this may affect the code of the clients that would "normally" not have been modified. This can have important effects on the maintainability of the software. Keeping a good separation between the interactional elements and the rest of the component would be better.

3.3 A new way to design the interactions in a distributed context

We have seen that mixing interactional pieces of code with the rest of the component code can lead to problems. To solve them, we propose moving the interactional pieces of code out of the functional component and putting them into a logical interaction unit, as shown in figure 5.

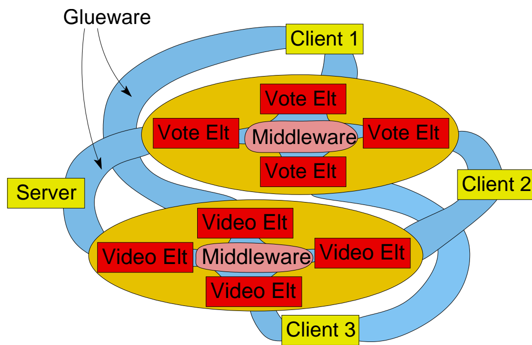

Figure 5. Reorganization of the interaction elements

For instance, all the vote elements belong to a single logical unit physically distinct from the server and the client components. All these elements communicate among themselves through middleware as before. The server component has a reference on a single element that is no longer an internal part. Now all the elements concerning a given interaction have been removed from the components and are logically coherent and reunified in a single entity.

We consider that all these parts of a given interaction form a logical software component. It is a special component because it is dedicated to managing the interaction, the communication or the collaboration among other components. For this reason, we call these kinds of components *interaction components* (or *mediums* in order to differentiate them from "classical" functional components). A medium integrates or reifies an *interaction abstraction*[2].

[2]Abstraction is used in the sense that "the details are hidden" and not "fuzzy"

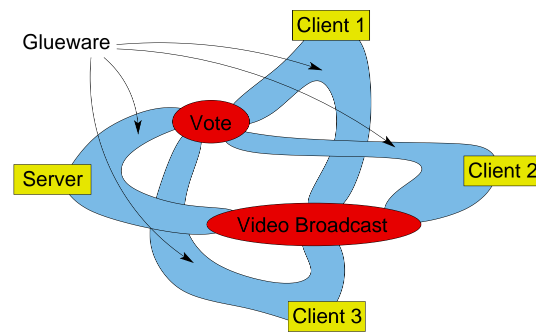

Figure 6. New global architecture of the application

If we represent the interaction abstractions as mediums in the previous application, this gives figure 6. We retrieve the four functional components (the server and the three clients) and the two mediums (the vote and the video broadcast). The communication between the components is entirely managed by the mediums. The components are connected to mediums via some glue used to adapt the medium services to the needs of the component that call them (the glue that links a component to a medium plays a role of adapter as could occur in a classical connection between two components). Since a medium is a component, it offers services like for any conventional component to components that are connected to the medium.

Compared to a classical architecture at the implementation level, figure 6 focuses not only on the functional concern but also on the interactional one. All the interactional concern has been removed from the components and now becomes visible as mediums, i.e., this concern is well decoupled from the others.

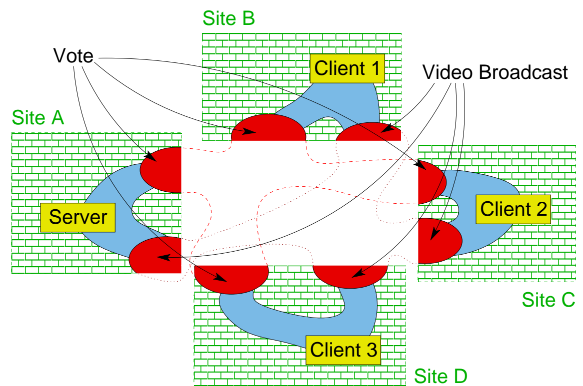

Figure 7. Deployment architecture of the application

The deployment of the application is shown in figure 7. Each server or client is deployed alone on a given site. We can notice that the mediums are split and a part of each medium is deployed on the same site as the component it is associated with. This deployment architecture is the consequence of moving the code shown in figure 5.

One very interesting point is that a component has a local reference on a medium. Since all the communication inside the application is managed by the mediums, the components no longer need to communicate directly among themselves (by the way of remote method calls for instance). The problems of localization and distribution of the components are entirely managed inside the mediums, as we shall see in the following sections.

4 Definition of interaction components

We have seen that an interaction can be implemented in a software component, an interaction component or medium.

An interaction component is the reification of an interaction, communication or coordination system, protocol or service into a software component. These interaction protocols or systems implemented or integrated in mediums are various in type and complexity: an event broadcast, a consensus protocol, coordination through shared memory (like Linda[8]), a multimedia stream broadcast or a vote system, for instance.

A distributed application is then built by interconnecting "conventional" components with mediums that manage their communication and distribution.

The reservation interaction (specified by a UML collaboration) of section 2 can thus be implemented with a medium. Figure 2 represents an application containing four components that will be deployed and interconnected among each other to realize the application: the functional airline and travel agencies components and the "interactional" reservation system component (that we can call the reservation medium).

An interaction component is first of all a component. Although the component paradigm is now widely accepted by the software community, there is no real consensus on the definition of a component. However, we can summarize its principle properties [16, 5]:

- It is an autonomous and deployable software entity.
- It clearly specifies the services it offers and those it requires. This allows the use of a component without knowing how it works (by looking at the code for instance).
- It can be combined with other components.

A medium thus has all these properties and, moreover, it is especially designed to be reusable. It offers services that can be easily adapted depending on the context of use.

In an application, the interaction or communication concern is managed by the mediums and the functional concern is localized in the classical components. This allows a good separation between these two concerns.

4.1 Specification of an interaction component

A medium is the reification of an interaction or collaboration. In UML, collaboration diagrams allow interactions among classes or instances to be described. A collaboration is thus suitable for specifying a medium. At the specification level, a medium is specified by a collaboration. This leads to a very interesting feature: if a collaboration represents an interaction between components, this collaboration can be implemented by a medium. Thus, a UML collaboration can be reified into a medium at the implementation level, ensuring a good traceability from the interaction specification to its implementation.

Depending on their needs, components use some services of the medium, but not all of them. For instance, in a broadcast medium, a component wanting to send information only uses a broadcast service. On the other hand, a component wanting to receive information uses a receive service. Components are differentiated depending on the role they play from the medium point of view. For example, a broadcast medium defines a sender and a receiver role. These roles match the roles used in UML collaborations. Since a medium specification is a collaboration, components connecting to a medium play a given role in this collaboration. With each role is associated a list of offered and required services.

The design of a collaboration must deal with the component characteristics: the interfaces of the services offered and required must be present. A special class inside the collaboration represents the medium as a whole. Like for any conventional components, mediums require a good specification in order to make them easily usable. A "good" specification includes all the information that describes how to use the component and also what it does. This could be encapsulated in a contract as we propose in [2]. This contract must include the required and offered service signatures, but cannot be limited to these. The semantics and the dynamic behavior of services must also be specified. This contract is the usage contract of [4].

In order to be able to completely specify a medium in UML, the collaboration diagram is augmented with OCL constraints, statecharts or any useful kind of UML diagram.

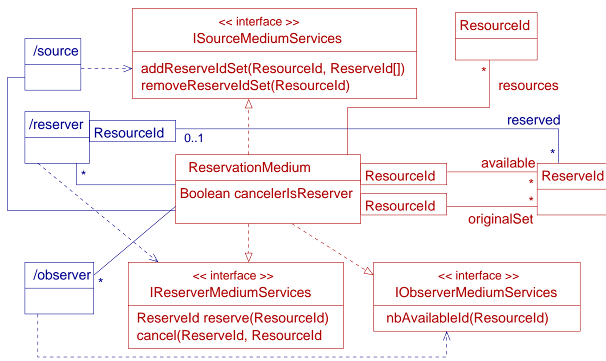

Figure 8. Collaboration describing the reservation medium

4.2 Specification of the reservation medium

The UML collaboration used in the instance diagrams in figures 1 and 2 represents an interaction through a system of identifier reservations. This interaction can be reified into a medium. Thus, the collaboration has a special design according to the specification principles we have outlined above. The structural view of the collaboration[3] is shown in figure 8.

The `Reservation Medium` class represents the medium as a component. It implements interfaces of services, each one being associated with a role. For instance, components playing the reserver role can call the service `reserve` or `cancel` of the `IReserverMediumServices` interface in order to reserve an identifier and to cancel a reservation.

The medium manages the reservations of identifiers for several resources (e.g., planes or buses). Each resource is identified by an instance of the `ResourceId` class and an identifier is an instance of the `ReserveId` class. For a given resource, the medium has a reference on the set of available identifiers (`available` link qualified by the `resourceId` value associated with the resource). This set is a subset of the `originalSet` link of this resource. Each service offered by the medium is specified by OCL pre and post-conditions. They allow the behavior of each service (i.e. the modification of the reservation sets) to be defined.

The methodology of specification and complete examples of medium specifications (including the reservation medium) are described in [3] and on our web site[4].

4.3 Architecture of an interaction component

We have seen in section 3 that mediums can be a better way to organize the code realizing an interaction. This section explains that the resulting architecture also has other advantages.

A medium implements an interaction system, protocol or service. Conventional functional components interact among themselves by using one or more mediums. The components are generally distributed over a network. In a classical implementation (i.e., without mediums), the components will directly communicate with others by means of remote method calls. Since the mediums now integrate these interactions, no more direct communication among the components is needed. A component just has to call a local service on the medium (directly on it or via some glue) to realize the interaction. So the component using a medium does not need to know the localization (on the network) of the other components that participate with it in the interaction. This localization problem is managed by the medium.

To be able to offer a local service to a component and to manage the localization of the components using it, a medium must be composed of several elements. Each of them is associated with a component, and both are deployed on the same site. The component has a local reference on this element on which the required medium services are called. All these elements are then distributed over a network and communicate among themselves through middleware. This architecture is shown in figure 9. These elements are called “role managers”. The type and the goal of a manager depends on the role played by the component connected to the medium. For instance, the component `TravelAgencyA` has a reference on a `Reserver Manager` whose responsibility is to manage the medium services from the point of view of a reserver component (here the component `TravelAgencyA`).

A medium differs in one point from a classical component that is a single entity deployable on a given site: it is composed of distributed elements that communicate among themselves. The single entity is only logical and not physical at the deployment level.

This implementation architecture allows a very good separation between the functional and interactional parts of an application at the implementation and deployment level. These parts are not mixed as is usually done. Interactional parts are completely localized in the managers. A

[3] Collaborations are generally described at instance level. In this paper, all the collaborations are described at specification (or class) level (see [11, page 3-109]) because we want to specify generic interactions and not application dedicated ones.

For some readers, the collaboration in figure 8 may appear somewhat “strange” because there are no messages (i.e. operation calls) on it. But this is a collaboration diagram and not a class diagram. Actually, we do not need to show these messages in this particular collaboration because OCL specifications are sufficient for defining the medium services.

[4] `http://www-info.enst-bretagne.fr/medium/` section “Specification in UML”. The reservation medium specified on our site manages only a single resource and is therefore slightly different from the one described above.

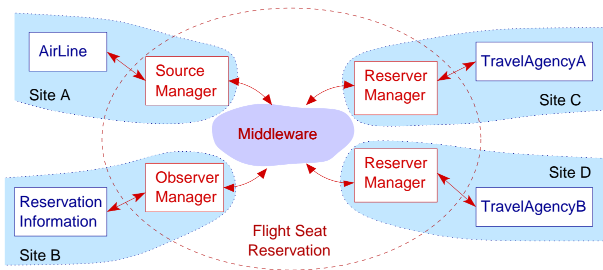

Figure 9. Architecture detail of the flight seat reservation application

medium can then be completely developed independently of any context of use and then be reused in different contexts.

Moreover, a component connected to a medium will only have to call its services offered. No assumptions about the way these services are implemented have to be made. This allows different implementations of the same services. And very different implementations can be made because of an important feature of our implementation architecture: the managers can be as complex as desired (they can have a state or manage data for instance). If they had only played a limited proxy role (such as Corba stubs and skeletons), this would have considerably reduced the number and kinds of possible implementations.

5 Using interaction components to improve non-functional constraint management

In the previous sections, we have seen how to specify and how to architecture a medium. The specification was made at the highest-level, i.e. at analysis level. Indeed, it focuses only on the usage contract of the medium. This contract is independent of any context, it only describes what the medium does and not how.

In order to be able to specify an implementation adapted to implementation constraints or requirements, we have developed a refinement process. This process transforms the specification at the analysis level into a low-level one corresponding to an implementation design including these implementation constraints.

Without detailing the process completely, its principle properties are the following:

- The process is composed of several steps. Each step modifies the complete specification (collaboration diagram, OCL constraints and statecharts) of the preceding one. Some steps can be entirely or partially executed in an automatic way by a UML case tool supporting UML model transformations such as Umlaut[7]. Others must be done manually by the software designer, notably those in which the implementation constraints and choices are involved.

- The goal of the process is to define implementation designs and deployment choices. One single specification at analysis level can lead to several implementation designs. These designs can be considered as the realization contracts[4] of the medium. These realization contracts must respect the usage contract.

- The implementation specification matches the architectural implementation design seen in section 4.3. In the collaboration diagram, the medium as a whole, as a single entity (represented by the class called `ReservationMedium` in our example) is replaced by a set of "role manager" classes. They have the responsibility of implementing the services of the medium. Each role (i.e., each component connected to the medium) is associated with a role manager. Some special managers that are not associated with a component can be used if needed.

We will present three different implementation designs for the reservation medium we have previously seen. These designs are the final results of the refinement process. We focus only on the lowest level without explaining the different specification transformations made throughout the process[5].

5.1 A basic implementation design

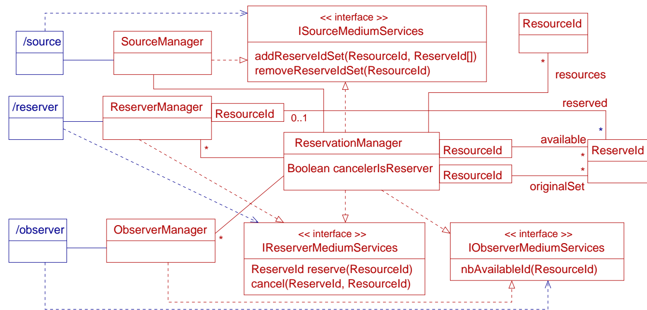

Figure 10. Implementation choice: centralized data set management

A first and simple implementation is to give the responsibility for managing all the sets to a new single manager. Figure 10 shows the collaboration design of this

[5] The process and the complete transformations of the reservation medium are available on our web site: `http://www-info.enst-bretagne.fr/medium/`, section "specification in UML"

choice that is the refinement of the collaboration at analysis level (figure 8). The class `ReservationMedium` has disappeared and has been replaced by the set of different kinds of managers: `ReserverManager`, `ObserverManager`, `SourceManager` and `ReservationManager` (the new special manager) classes. The role managers (observer, reserver and source managers) are kinds of proxies that just relay the queries of their component to the reservation manager that entirely manages all the sets. This manager is special because it is not associated with a component.

The OCL constraints are also modified by the refinement process. Some methods, services and OCL constraints will also be added in order to specify the parts specific to this implementation choice.

The interfaces of services have not been changed during the process. The services offered and their semantics remain the same. The interaction specified is still the same from the point of view of its users.

This basic implementation design is fairly easy to implement because all the complexity of the interaction (set management) is located on a single site. But, as we have seen, this implementation cannot be used in all contexts, especially if the number of connected components is too high; this single manager will be a bottleneck. However, this implementation design is very suitable for small applications in terms of interacting components and the number of resources to manage, like the bus seat reservation application.

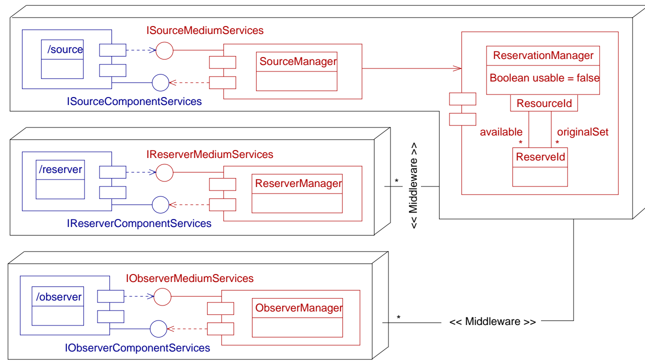

Figure 11. Deployment choice for the centralized version

Figure 11 shows a deployment diagram of this implementation. The choice made here is to deploy the reservation manager on the same site as the source manager.

5.2 A more complex implementation design

In this design (see figure 12), the implementation choices include a distributed management of the sets of identifiers. Each reserver manager possesses a part of each identifier set (the qualified link `originalLocalSet`). Centralized management of the sets is no longer required. This design will suit the requirements of the large flight seat reservation application: each travel agency will have a small set of identifiers (managed by the reserver manager it is associated with) in which the reservations of clients will be made. This obviates the need to communicate with a single remote server that could be a bottleneck.

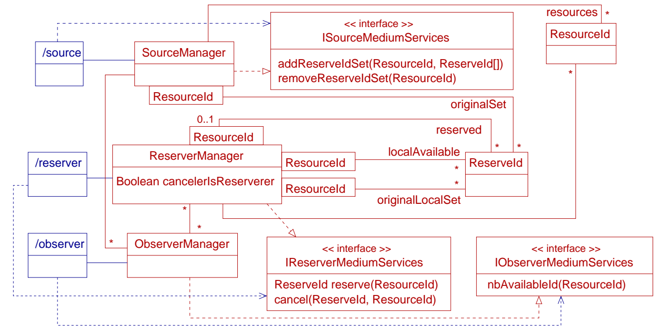

Figure 12. Implementation choice: distributed data set management

Figure 13 shows a deployment choice for this implementation design in which we can see that the sets are completely distributed.

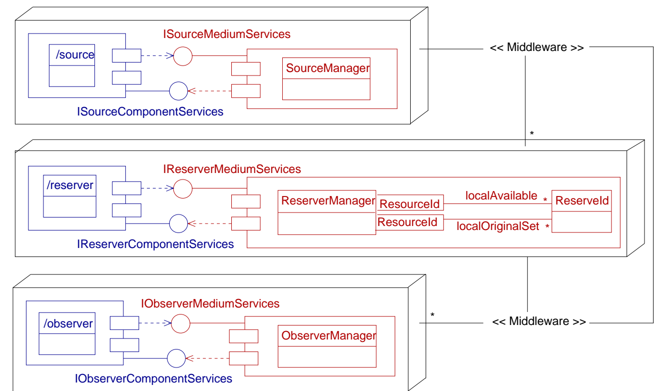

Figure 13. Deployment choice for the distributed version

5.3 A third implementation choice

A third design could be an intermediate one between the first two. Indeed, the complete distribution of the sets implies using very complex algorithms to allocate (and reallo-

cate if a local set is empty) the identifiers according to the need of reserver managers. The centralized version, however, is not able to stand the augmentation of load.

An interesting compromise between load resistance and easy implementation and maintainability, is to have several reservation managers instead of just one. Each of them has the responsibility of managing only a small part of the resources and not all of them. The reserver managers do not directly manage reservation identifiers as they do in the distributed version. This allows a distribution of the managers' queries on several sites and thus the workload. Depending on the resource, a reserver manager will send its query to the relevant reservation manager.

In the context of the flight seat reservation application, we could imagine several specializations of the reservation managers: one manager could manage only national flights, another one the international ones. Or this could be based more on deployment: a manager could deal with a particular country or a precise geographical area.

5.4 Non-functional constraints management improvement

Of course other implementation designs can be found. However, the above three show that it is relatively easy to specify (and then to implement thanks to our deployment architecture) very different implementation designs that are able to fit different non-functional requirements without changing the semantics of the services offered. Here we have focused on the number of interacting components and resources to handle, but other non-functional features can also be managed.

An important aspect of this process is that the different specifications and implementations are done completely independently of any context of use or any application. A medium is then completely reusable and manipulable at all levels of application development. A software designer can study the specification of a medium at the analysis level to know the services it offers. Then he can use a lower-level medium specification in his application design, according to the design and implementation choices he desires or non-functional constraints he has to face. Finally, depending on his deployment constraints, he can choose the adapted deployment specification and use the according implementation.

6 Related works

6.1 Architecture Description Languages

A wide field of software engineering concerns software architecture [1]. In order to improve maintainability, evolutivity and reusability, many Architecture Description Languages (ADL) have been defined [10].

Like for most ADLs, we consider connectors (mediums) as first-class entities, but the main differences between the ADL approach and ours are:

- Most ADLs allow a hierarchical decomposition of components while our communication components have (until now) a flat structure.
- We propose a full refinement process (not entirely described in this paper) from the discovering of communication abstraction at the analysis stage to the implementation.
- We ignore the actual underneath communication services (such as RPC, Unix pipes, http-link, etc.) until implementation.

This last point is essential. Mediums make a clear distinction between the communication abstraction and its various implementations, making medium substitution a good way to manage system evolution.

6.2 Coordination components and languages

The separation between the objects' essence and their integration in an application is not always carried out well. Frequently, the realization of a class mixes both the objects' essence and interaction requirements in a particular application. In order to improve the separation of these two concerns, some authors propose object connectors [6, 13], which contain the glue required to make objects interact. This approach, like ours, reifies interactions that are usually described with a coordination or collaboration language.

In some implementations the refinement process consists of a compilation, which is more abstract, but prevents implementation variants; connectors are dedicated to a specific application making them potentially less reusable than if they were developed from *standard* interactions such as we propose. Another approach uses collaboration contracts [9] which are more flexible since collaboration rules can be dynamically updated.

Most coordination models rely on an explicit interface of components being coordinated [8, 9] but some use introspection and metaprogramming features to coordinate components that hide their interface [6].

However, we consider the coordination problem as a subproblem of the communication, limited to message passing models[6]. Until now, we have not specified a medium dedicated to collaboration but we imagine it would be possible;

[6] Even Linda which emulates asynchronous message passing through a shared memory.

a collaboration medium would be programmed using a coordination language and would observe events on its entries and trigger events as specified.

6.3 Catalysis

Catalysis [5] is a methodology that is component centered. It uses a notation based on UML to describe models, components and implementations. The concept of collaboration is central and defined as follows[5, page 23]:

> A *collaboration* is a collection of actions and the types of objects that participate in them.{...} Catalysis treats collaboration as first-class units of design work.{...} Collaborations can be generalized and applied in many contexts.

Moreover, Catalysis proposes to define collaboration frameworks that are kinds of generic collaborations. Catalysis collaboration frameworks resemble our medium specifications since a catalysis connector is specified by a collaboration framework (a medium being a kind of connector). But Catalysis proposes to define only a small set of connectors that can be used during the implementation. We believe that communication abstractions between distributed components cannot be limited to a small set of low-level interaction patterns, even at the implementation level. We argue that every communication abstraction can be manipulated as a connector or a medium, independently of its complexity.

Finally, although Catalysis offers a methodology to refine its models and to keep track of the successive refinement steps, and although collaborations can be refined in connectors, Catalysis proposes no real process nor any implementation target as we do. We imagine the work presented here as being an extension of Catalysis methodology.

7 Conclusion

We have presented an architecture and a framework to specify and implement interaction abstractions in distributed contexts. The solution we propose is based on the reification of interactions into components (mediums) both at specification and implementation level allowing the separation of functional and interactional concerns throughout the software development process.

At the specification level, a medium is specified with a UML collaboration (augmented with OCL constraints) that follows specific rules in order to suit component specification requirements. Then a refinement process transforms this specification into a low-level implementation and deployment design. The process can lead to various implementations depending on non-functional constraints.

Indeed, with the study of real-life applications, we have shown that the implementation of the same interaction or collaboration can be highly dependent on non-functional features (from the point of view of the "interaction semantics"), such as the localization and number of interacting components, the state of the network used, the need to encrypt communication or the need to have a fault-tolerant system, for instance. In our examples, we have focused on scalability that involves using very different implementations of the same interaction depending on the size of the application.

Then, we have presented a deployment architecture of interaction components allowing both the separation of concerns at the implementation level and facilities to implement variants of the same interaction: a medium is composed of distributed elements that are logically coherent among themselves. Each element is associated with a component and has the responsibility of realizing the services of the medium this component requires. As these mediums' parts can be as complex as desired, various implementations of the same interaction are possible. We have also written a Java framework for implementing mediums and the interactive video application described in this paper[7]. This has shown the benefits of having complex interaction abstractions through mediums at the implementation level.

An important aspect of this complete process is that the different specifications and implementations are realized completely independently of any context of use or any application. A medium is then completely reusable and manipulable at all levels of application development. A software designer can study the specification of a medium at the analysis level to know the services it offers. Then he can use a lower-level medium specification in his application design, according to the design and implementation choices he desires or non-functional constraints he has to face. Finally, depending on his deployment constraints, he can choose the adapted deployment specification and use the appropriate implementation.

In the future, we plan to build a comprehensive catalogue of interaction components including UML specification and implementation variants according to non-functional constraints. The UML specification methodology and the refinement process could be integrated into an UML CASE Tool such as Umlaut[7]. This will help software designers to build applications by reusing interaction components and to define their own mediums with their variants.

[7]Readers interested on this framework are invited to consult our web site: `http://www-info.enst-bretagne.fr/medium/framework/`. The framework and applications using it are freely downloadable and distributed under the GNU GPL.

References

[1] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, pages 38–45, July 1999.

[3] E. Cariou and A. Beugnard. Specification of Communication Components in UML. In H. Arabnia, editor, *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '2000)*, volume 2, pages 785–792. CSREA Press, June 2000.

[4] J. Cheesman and J. Daniels. *UML Components - A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2000.

[5] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[6] M. Günter. Explicit connectors for coordination of active objects. Master's thesis, University of Berne, 1998.

[7] W.-M. Ho, J.-M. Jézéquel, A. Le Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, Oct. 1999.

[8] T. Kielmann. Designing a Coordination Model for Open Systems. In S. Verlag, editor, *Coordination Languages and Models*, Lecture Notes in Computer Science 1061, 1996.

[9] L.Andrade, J.Fiadeiro, J.Gouveia, A.Lopes, and M.Wermelinger. Patterns for coordination. In G.Catalin-Roman and A.Porto, editors, *Coordination Languages and Models*, pages 317–322. LNCS 1906, Springer-Verlag, 2000.

[10] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, 1997.

[11] OMG. Unified Modeling Language Specification, version 1. 3. `http://www.omg.org`, jun 1999.

[12] T. Reenskaug. *Working with Objects*. Manning/Prentice Hall, 1996.

[13] M. Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In D. Lamb, editor, *Studies of Software Design, Proceedings of a 1993 Workshop*. Lecture Notes in Computer Science 1078, Springer-Verlag, pp. 17-32, 1996.

[14] J. Siegel. *CORBA Fundamentals and Programming*. Wiley Computer Publishing, 1996.

[15] G. Sunyé, A. Le Guennec, and J.-M. Jézéquel. Design pattern application in UML. In E. Bertino, editor, *ECOOP'2000 proceedings*, number 1850, pages 44–62. Lecture Notes in Computer Science, Springer Verlag, June 2000.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.

[17] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.