# An Implementation Method for Distributed Object-Oriented Applications

Vander Alves[*] Paulo Borba [†]
Centro de Informática
Universidade Federal de Pernambuco

***Abstract***

*Distribution has become an essential non-functional requirement of most applications. The same application may be required to use different distribution platforms simultaneously or during its evolution. However, distributed applications are considerably more difficult to design, implement and test than centralized systems even with the same functionality. We present an implementation method which guides the progressive transformation of an initially centralized application into a distributed one. The method helps to tame the inherent complexity of distributed systems and makes tests more effective. Additionally, it relies on a design pattern promoting extensibility and reuse of other layers such as the user interface and the business layers.*

## 1 Introduction

Over the last decade, with the rapid growth of the Internet and the World Wide Web, distribution has become an essential non-functional requirement of most applications, spanning areas such as Electronic Commerce, Distance Learning, and Artificial Intelligence. In fact, distributed applications may have a number of advantages over centralized systems: scalability, fault tolerance, and resource sharing.

However, despite these advantages, the design and test of distributed applications are considerably more difficult than those of centralized systems. Indeed, problems such as non-determinism, *livelock*, and *deadlock* are easily introduced even in distributed systems containing few components. The detection of these problems is also harder since centralized system testing techniques do not apply effectively in the distributed scenario. Moreover, distributed applications are commonly implemented with other non-functional requirements such as concurrency and persistence, which make their implementation a complex task.

A great variety of languages and tools has been developed to support the implementation of distributed systems. For instance, Java [7] has a distributed object model provided by RMI [16], which supports the implementation of distributed object collaborations in the language itself without the need for low-level protocols. In addition, RMI also provides a tool for automatic generation of components involved in the development of distributed applications.

However, languages and tool support are not sufficient to avoid the problems which arise in developing distributed applications. Support at a higher level is also required. In

[*]Supported in part by CNPq. Electronic mail: `vra@cin.ufpe.br`. Av. Professor Luis Freire s/n Cidade Universitária 50740-540 Recife PE Brazil.

[†]Supported in part by CNPq, grant 521994/96–9. WWW: `http://www.cin.ufpe.br/~phmb`. Electronic mail: `phmb@cin.ufpe.br`.

this context, the design of the software architecture [14] of such applications is critical. The architecture describes the system's overall organization, its components, and how they interact. It helps to tame the system's inherent complexity and provides a partial blueprint for its construction. Furthermore, it promotes software's reusability and extensibility.

In order to implement a software architecture, its components and their relationships must be refined. Patterns [5, 6, 13] play a central role in this. By using them, programmers build software systems with predictable non-functional properties. Therefore, a pattern supporting software extensibility leads to the construction of applications where a change in a non-functional requirement, such as persistence or distribution, does not entail the need to redevelop most of the system.

On the other hand, current software development processes, despite improving analysis and design methods, do not emphasize implementation methods [8]. In fact, implementation is commonly handled in an *ad hoc* manner. As a consequence, the benefits of even a carefully designed architecture may be lost: although the resulting code may be object-oriented, it can still tangle aspects of different layers and thus is difficult to reuse and extend. This risk is specially higher when implementing distributed, concurrent, and persistent applications, which account for a significant part of Web-based information systems.

In this context, this work presents and evaluates an implementation method for introducing distribution into an application. Our goal is to improve software quality, by considering reuse and extensibility, and to increase development productivity, by considering the time spent during implementation. As an implementation method, the one described in this work assumes that critical properties of a distributed application (such as scalability, transaction support, and security) have already been considered in the design of the application. Such design issues are preserved by the implementation method we present.

The rest of this paper is organized as follows. Section 2 overviews a design pattern for distributed applications. An implementation method relying on this pattern is presented in Section 3. An experiment in which the method was applied is described and analyzed in Section 4. Related approaches are considered in Section 5, and Section 6 summarizes our contributions.

## 2 A design pattern for distributed applications

This section overviews the Distributed Adapters Pattern (DAP) [1], which we have developed with the purpose of refining the distribution layer of distributed architectures. It is actually a combination of the Facade, the Adapter, and the Factory design patterns [6].

Indeed, well known patterns for structuring distributed systems already exist. The Broker [5] and the Trader [5] patterns are two examples. These are architectural patterns [5] and focus mostly on providing fundamental distribution issues, such as marshalling and message protocols. Therefore, they are mostly tailored to the implementation of distributed platforms, such as CORBA [11]. DAP uses these fundamental patterns and provides a higher level of abstraction: distribution Application Programming Interface (API) transparency to both clients and servers. Section 5 compares DAP to other patterns.

DAP introduces a pair of object adapters [6] to achieve better decoupling of components in distributed architectures. The adapters basically encapsulate the API that is necessary for allowing distributed or remote access of business objects. In this way, the business layer of an application becomes independent with respect to the distribution layer, so that changes in the latter do not impact the former.

There are two kinds of adapters: **source adapters** and **target adapters**. Roughly, the latter wraps server business objects in the places where they are located, and the former represents those objects in remote locations. In a typical interaction, a user interface object (a GUI, for instance) in one machine would request the services of a source adapter located in the same machine. The source adapter would then request the services of a corresponding target adapter residing in a remote machine. Finally, the target adapter would request the services of a Facade object co-located with the target adapter. Figure 1 illustrates this example.

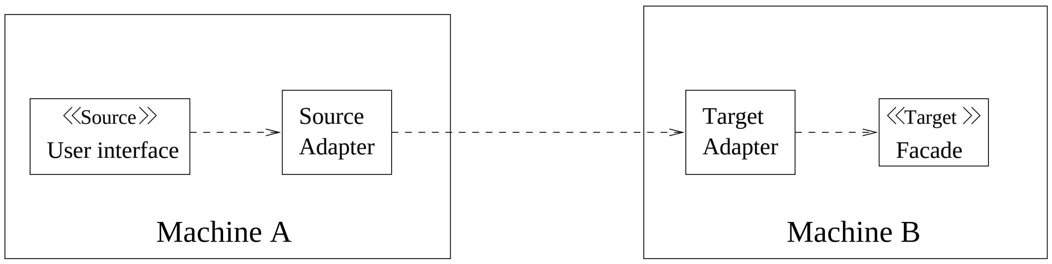

Figure 1: An example of DAP.

The Facade component in Figure 1 is related to business aspects of an application. For instance, the Facade of a banking application keeps references to entities such as account and customer records, and has operations for manipulating these entities:

```
class Bank implements IBank {
   private AccountRecord accounts;
   Bank() throws BankInitializationException {...}
   int addAccount(Account account)
             throws AccountAlreadyExistsException {
          accounts.add(account);
   }
   void deposit(String accountNumber, double value)
             throws UnknownAccountException {
         accounts.deposit(accountNumber,value);
   }
}
```

where `AccountRecord` provides services for manipulating a record of accounts (insertion, updating, querying, deletion, etc.) and also for depositing into or withdrawing from them. The exceptions `AccountAlreadyExistsException` and `UnknownAccountException` are specific to the banking application. The `IBank` interface implemented by the banking facade is a business **facade interface**. It abstracts the behavior of the application:

```
interface IBank {
   int addAccount(Account account) throws CommunicationException,
             AccountAlreadyExistsException;
   void deposit(String accountNumber, double value)
          throws CommunicationException, UnknownAccountException;
}
```

where `CommunicationException` is a general exception representing failure in the distribution layer. It does not depend on any particular distribution technology and is defined since the application will eventually become distributed.

Source and target adapters provide a higher level of abstraction than stub and skeletons do. The adapters isolate user interface and business code from distribution API, whereas stubs and skeletons isolate user interface and business code from the implementation of distribution issues, but not from distribution API. Source adapters delegate lower level distribution issues such as marshalling to stubs, and target adapters delegate such issues to skeletons.

## 2.1 Consequences

DAP provides the following benefits:

- *Modularity.* This pattern structures distribution aspects modularly, promoting loose coupling between the different layers of an application's architecture: distribution, business, and user interface layers.

- *Reuse and extensibility.* Due to the modularity provided by the pattern, developers can reuse business components easily in other applications based on other middleware technologies. In addition, changes to the middleware aspects are simpler, since these are restricted to the distribution layer.

- *Progressive implementation.* The pattern supports progressive implementation. During the early phases of the development, developers construct a functionally complete prototype, where a GUI, for example, depends directly on a business Facade. Later, developers add the distribution layer seamlessly, since this layer implements the same interface as the Facade.

This pattern has the following drawbacks:

- *Increased number of classes.* A pair of adapters, three factories, and an initializer are necessary; however, their structure is simple and their generation could be mostly automated by tools.

- *Extra indirection.* The pair of adapters introduce two additional method calls for the same remote request. However, both of these are local calls, which are much less expensive than remote ones. The work in [1] shows empirical data analyzing the impact on efficiency caused by the adapters; the analysis reveals that such impact is minimum.

## 3 Method

This section presents an implementation method that guides the developer in the process of turning an initially non-distributed application into a distributed one, structured according to DAP. Although we describe and illustrate the method in the context of distributing one application, the method is more general: it is useful for distributing any kind of service remotely available. Since an implementation method is described (rather than a design method), it is assumed that issues such as scalability, transaction support, and security have already been considered in the design of the application. The method presented here preserves such design issues.

The method consists of five steps, each one describing the implementation of the components of the design pattern mentioned in Section 2. The activities should be executed sequentially. These activities are illustrated with RMI [16]. However, developers may follow these steps when using other platforms, such as Jini [17] and CORBA [11], for example, with minor modifications. The following subsections describe the method's steps.

### 3.1 Preliminary step: prepare facade and client

The implementation method described in this work refines the Progressive Implementation Method (Pim) [4], thereby relying on its architectural and design patterns. The goal of this step is to guarantee that these patterns are used in the initially non-distributed application. Actually, this step should already have been considered in the design activities and in the implementation of such application. Therefore, we list it here just as an optional corrective action for possible failure in these previous activities. The step described in this section is itself divided into three parts, which we list separately.

#### Different parameter passing mechanisms

In order to explain this step, let us consider the following banking facade interface:

```
interface IBank {
   Iterator getAccountsList() throws CommunicationException;
   Iterator getClientsList() throws CommunicationException;...
}
```

where `Iterator` is a type with methods for navigating through a sequence of items. The design process must have identified the parameter passing mechanism of the types in the facade interface. Since this interface describes the distributed behavior of the application, the parameter passing mechanisms are also described in this context. Therefore, in the banking interface above, `Iterator` could be passed by value in `getClientsList`, in which case an object of this type would be serialized to be sent from a server to a client, and by reference in `getAccountsList`, in which case a proxy would be sent from the server to the client.

In the latter case, the semantics of interaction with an `Iterator` object is affected by distribution: communication errors may arise, for instance; therefore, methods in `Iterator` must declare an exception like `CommunicationException` to indicate this enhanced semantics. In addition, clients of this type must treat this exception. This implies that clients of `getClientsList` also have to treat this exception. However, this latter handling is unnecessary and conceptually inadequate, since in this case the client receives an `Iterator` object through serialization, and thus this object is a local and not a distributed one. To avoid this, we may create a new type, `RemoteIterator`, rename the return type of `getAccountsList` to this new type, and update references to this new type in clients of this method. The facade interface thus becomes

```
interface IBank {
   RemoteIterator  getAccountsList() throws CommunicationException;
   Iterator getClientsList() throws CommunicationException;...
}
```

where `RemoteIterator` and `Iterator` are separate types for each parameter passing mechanism:

```
interface Iterator {
   Object next();...
}
interface RemoteIterator {
   Object next() throws CommunicationException;...
}
```

With these, clients of `getClientsList` handle local objects explicitly and clients of `getAccountsList` manipulate actual remote objects.

To summarize, in this step, the developer should perform the following actions:

1. Identify types in facade interface **f** that are used with different parameter passing mechanisms.
2. For each such type **t** create a new one, **t'**, with the same methods of **t** but with all methods declaring `CommunicationException` in their `throws` clause.
3. Replace **t** with **t'** in **f** wherever **t** is a return or a parameter type whose arguments are passed by reference.
4. Guarantee that methods in **t** do not declare `CommunicationException` in their `throws` clause.
5. Update clients of the methods of **f** passing **t** by reference so that these clients now refer to **t'**.

### Keep the client consistent

Suppose that in our banking application there is an operation for adding accounts: this operation adds an account into the server and returns the account number to the client. The operation itself generates a number, which is not informed by the client at the method call. A careless specification of the facade interface would be:

```
interface IBank {
   void addAccount(Account account) throws CommunicationException,
             AccountAlreadyExistsException;
}
```

This specification assumes that the client would be following this protocol: the client could create an `Account` object, call `addAccount`, passing this new `Account` object, and then retrieve the account number by using an accessor method of `Account`, such as `getAccountNumber`. This protocol would work in the initially non-distributed version of the application, since there would be only one copy of `Account`. However, in the distributed version of the application, there would be two copies of `Account`: one at the client and one at the server; the copy whose account number is set is the one at the server. Therefore, the protocol above would not work since the client would be retrieving the account number of the `Account` copy at the client, which is inconsistent with the one at the server.

The problem above could be avoided by explicitly returning the account number in the operation:

```
interface IBank {
   int addAccount(Account account) throws CommunicationException,
               AccountAlreadyExistsException;
}
```

This specification would prevent the client from following the erroneous protocol mentioned above. In general, we should

1. Make all methods in the facade interface return values if they should do so conceptually.
2. For these methods, propagate implementation changes to the corresponding methods in the business and data layers.
3. Change clients of these methods so that they use the values returned explicitly in the methods.

### Keep the server consistent

When interacting with the server of an application, a client may retrieve business objects, such as `Customer` objects, and set its attributes. After such interaction, the client must in addition remember to call an update method in the facade interface in order to keep those business objects at the server consistent with the corresponding copies at the client.

For example, clients of the `IBank` facade interface typically write code such as the following:

```
customer = bank.fetchCustomer(customerId);
```

for fetching a business object, where `customer` is either an attribute of a class in the client (a GUI, for instance) or a local variable in the `doPost` method of a servlet [19]. The `fetchCustomer` method would be called when an event happened at the GUI or when the servlet received a HTTP request, which could actually come from another GUI in a Web browser. Later the client could interact with `customer` and update its attributes:

```
customer.setName(...);
customer.setAddress(...);
```

where these invocations would be in a different method in the GUI or in another servlet. Actually, since servlets interact with HTTP, which is a stateless protocol, this servlet would have to call `fetchCustomer` again before `setName` and `setAddress`. However, in either case and in the distributed version of the application, `customer` will not hold a reference to the original customer; instead, it will hold a reference to a copy of the original one. Therefore, in order to reflect the changes in the original customer, we should add the following method call at the end of that code:

```
bank.updateCustomer(customer);
```

where `IBank`'s `updateCustomer` method is responsible for fetching and updating the original customer with the attribute values of its argument. Therefore, the actions in this step are:

1. Trace clients of the facade interface for updates to business objects.
2. After such updates, call the corresponding `update` method in the facade interface.

## 3.2 Step 1: implement adapters

This is the first concrete step in turning a non-distributed application into a distributed one. The step assumes that the non-distributed application is structured according to the Facade design pattern and that the implementation is to be based on the Distributed Adapters Pattern (Section 2). In particular, we assume that the skeleton code for this pattern has already been generated. In addition, the developer should also have already considered the corrective actions described in the preliminary step presented previously.

Before the execution of this step, the implemented components in the original application are the user interface and the facade. Essentially, what this step does it to add a pair of adapters between these components; Figure 1 shows the structure of the result. These adapters are the core elements of the design pattern mentioned in Section 2 and, in essence, encapsulate and hide distribution details. In summary, for the facade of the application, we create a pair of adapters, referred to as source adapter and target adapter. These may be used to connect a user interface to a facade of a system (*main service adapters*) or this facade to another facade, representing an auxiliary service (*auxiliary service adapters*).

### Source adapters

For the source adapter, the following tasks should be performed:

1. Make it implement the facade interface of the Distributed Adapters Pattern mentioned in Section 2.
2. Implement every method of the facade interface, by delegating the call to the target adapter, calling a method of the same name, but replacing specific distribution exceptions with the generic `CommunicationException`, which is declared in the `throws` clause of methods in the facade interface.

3. Implement a `connect` method which allows the source adapter to find a reference to the target adapter. This method may be called by the source adapter's constructor and by the clients of this adapter if the reference to the target adapter eventually becomes invalid.

#### Target adapters

For the target adapter, we should do the following:

1. Make it implement a specific distribution interface, by means of which the source adapter will interact with it. Such interface is essentially the facade interface with the `throws` clause of each method augmented with platform specific exceptions.

2. Implement every method of the interface above, by delegating the invocation to the facade component, calling a method of the same name. Differently from source adapters, which replace specific distribution exceptions with a generic communication exception, the target adapter performs no exception filtering, since these adapters are tied to a specific distribution platform.

3. Implement the target adapter's constructor by registering it as a remote object at a name service. This service is searched by the source adapter's `connect` method.

## 3.3 Step 2: implement other non-functional requirements

The adapters presented in the previous step deal with basic distribution details and hide these details from the business and the user interface code. The adapters may also handle additional non-functional behavior, which also should not affect the business and the user interface code. In this step, we illustrate how the adapters may perform some of this additional behavior, which might be useful for implementing distributed applications. The execution of this step consists of enhancing the implementation of these adapters. Section 4, which evaluates the method, explains the reason for implementing this enhanced behavior as a separate step.

#### Fault tolerance

The source adapters presented previously have no fault tolerant behavior: if there is a communication error or if the server is unavailable, they simply raise a communication exception. Nevertheless, source adapters can also implement fault tolerant behavior.

If a source adapters receives a remote exception when interacting with the target adapter, it may implement the policy of trying to contact the target adapter again a certain number of times, or trying to contact another target adapter, representing a spare service. This policy, being implemented by the source adapter, is hidden from its client, a GUI, for instance.

#### Caching

Some operations may return a considerable amount of data, of which only part is useful at any moment. Sending everything to the client at once is not desirable since it may have a negative impact on network performance. One solution is to send a cache with part of the required data and to transfer more data every time a fault happens.

A source adapter can implement this caching behavior. When a querying operation returns many entries, part of them are used to initialize a source adapter. The client of this adapter (a GUI, for instance) retrieves the entries from this adapter. When a fault happens in the source adapter, it contacts the target adapter to retrieve more entries. This caching behavior is implemented in the source adapter and is transparent to the GUI.

## 3.4 Step 3: serialize objects

In the non-distributed application, there is usually only one copy of each business object. When the application is distributed, these objects may have to be copied to different machines. To accomplish this, these objects must be serializable; in RMI [16], this means that they must implement the `Serializable` interface. A similar approach is used with other technologies. In this step, we present serialization issues in the context of the design pattern we mentioned in Section 2.

### Facade types

In the facade interface of all services to be distributed, the developer should identify all non-primitive return types or method parameters passed by value. All such types, and their subtypes, must be serializable. For example, consider the following interface:

```
interface IBank {
   int addAccount(Account account) throws CommunicationException,
         AccountAlreadyExistsException;
   Iterator getAccountsList() throws CommunicationException;
}
```

where `Iterator` is an interface and `Account` is a class. After identifying those two types, we should then modify `Account` so that it implements the `Serializable` interface. In addition, we should also look for the classes implementing `Iterator` and also make them implement `Serializable`.

### Type dependency

For a type to be serializable, all types on which it depends (through inheritance or clientship) must also be serializable. Primitive Java types are serializable by default [7]. Therefore, the hierarchy of non-primitive types that are attributes of the types identified in the previous step must also be serialized. For example, if `Account` has a `Transaction History` attribute, then we should make this attribute serializable. If `Transaction History` has any attribute of a non-primitive type, then we serialize those types as well, and so on.

### Transient information

Some business objects may contain sensitive information that we do not want to send to clients because it is either useless or not secure. For instance, such information could be database connections, query results, and file handles. To prevent such information from being sent, these should be declared as `transient` attributes [7] in order to guarantee that they will not be serializable.

### 3.5 Step 4: prepare environment

To distribute an application, we must choose at least one platform. For this platform, there must be some configuration information, such as the location of a server and names of specific adapters, for example. In order to incorporate these details into the the application modularly, we use a set of Factories [6] in the design pattern presented in Section 2. Essentially, these factories are used in the creation of the adapters illustrated in the pattern.

Supporting these factories, there is, both at the client and at the server, a configuration file which specifies whether or not the application is distributed, identifies the distribution platform to be used, and indicates the location of one or more servers.

## 4 Method evaluation

The implementation method described in the previous section was validated by an experiment. This section describes the experiment and evaluates the method, discussing its soundness and usefulness.

### 4.1 The experiment

In the experiment, we implemented a distributed and persistent Web based information system, that allows manipulation of business entity records. The name of the system and its actual code is not disclosed for commercial reasons. We started from a non-distributed application which was already running in a company; we then implemented two use cases [3] of this application from its design, and finally we distributed this implementation. The technologies used in the development were

- Java [7], which is proposed by Pim [4];
- HTML and Servlets [19], for user interface across the Web;
- Oracle [12] and JDBC [18] for persistent storage and database connection;
- RMI [16] as the distribution technology.

In fact, the system has two distribution levels, one handled by HTTP and the other by RMI. The deployment diagram [3] in Figure 2 illustrates the system's physical structure.

### 4.2 Implementation

The project development had a team of three programmers: **S**, **T**, and **V**, where **V** is the author of this work. The project used the RUPim [10] process, which is a combination of RUP [8] with Pim [4]. Essentially, this means the development consisted of a series of iterations, each of these refined a set of functionalities from requirement specification

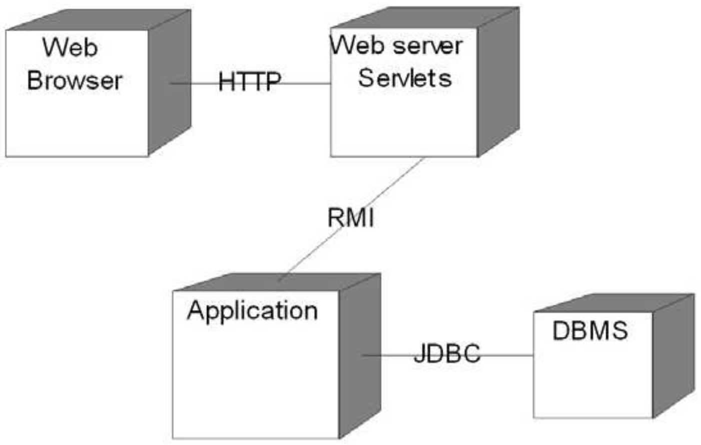

Figure 2: Physical structure of the implemented system.

through implementation. Former iterations considered major risks and were essential for establishing the software architecture. Latter iterations just added functionality to an already established architecture.

Initially, **T** and **V** built a functionally complete prototype; this prototype is a centralized and a volatile version of the system and implements only the business logic of the two use cases. In addition, these programmers also designed, but not implemented, distribution issues for this prototype. During the implementation of this version, the programmers performed unit and integration tests. After all of these, system tests were performed. Next, **V** implemented distribution aspects, and **T** implemented persistent aspects. The implementation of such aspects happened concurrently and independently; **S** implemented concurrent aspects later. After each such aspect implementation, the corresponding programmer performed integration and system tests.

In particular, **V** implemented distribution aspects according to the method described in Section 3. **V** implemented the initially non-distributed version of the application according to the conditions in the preliminary step (Section 3.1): different types for different parameter passing mechanisms were used, and the facade interface was changed to return data explicitly in one method. This change prompted modifications in some classes in the business and in the data layers.

Subsequently, in Step 1 (Section 3.2), **V** created a pair of main service adapters, the source adapter being used by the Web server and the target adapter by the server. In Step 2 (Section 3.3), a pair of auxiliary service adapters was implemented for caching in order to handle queries returning much data more efficiently. Later, in Step 3 (Section 3.4), **V** serialized four business classes and the source adapter of the auxiliary caching service. Finally, in Step 4 (Section 3.5), **V** created three factories. Table 1 summarizes the components implemented.

The implementation of all aspects consisted of 84 classes and 5.425 lines of code. Total implementation time was 33 hours. Table 2 shows how the implementation effort is fractioned considering several types of code.

| Component | Quantity Implemented |
|---|---|
| Main service adapters | 2 |
| Auxiliary service adapters | 2 |
| Serialized classes | 5 |
| Factories | 3 |
| Configuration files | 1 |

Table 1: Implemented components

| Types of code | Time (%) | Lines of code (%) |
|---|---|---|
| Business | 37 | 53 |
| Persistence | 33,5 | 37 |
| Distribution | 23,5 | 9,5 |
| Concurrency control | 6 | 0,5 |

Table 2: Fraction of implementation effort.

### 4.3 Discussion

Table 1 and the system's size refer to an implementation of two use cases. If more use cases were considered in the application, there would be a refinement in the implementation of the main service adapters, other auxiliary services would be created, and additional classes might be serialized. The number of factories and configuration files would not change. This behavior is expected for Web-based applications similar to the one in this experiment. For applications with additional levels of distribution (more levels than those depicted in Figure 2), there would be additional pairs of main and auxiliary service adapters. In addition, for each distribution platform used, there would be a configuration file and a specific distribution factory.

Table 2 reveals that, although distribution code represents a modest amount in the size of the implemented application, the time spent to implement such code was significant when compared to other aspects. In fact, this time could be reduced substantially since the adapters creation in Steps 1 and 2 was a time consuming and a tedious task; since the adapters structure is canonical, they could be generated automatically by tools. Aspects [9] may provide an alternative implementation, as discussed in Section 5.

The method is useful since it guides the developer in the non-trivial implementation of distribution aspects: there is a sequence of steps to follow, each one specifying the components to be implemented and where they should be deployed. Moreover, the method is sound, as it implements these aspects modularly, minimizing the impact on other layers of the application (business, and user interface, for instance). This has been useful to tame the inherent complexity of distributed applications.

In addition to helping to tame implementation complexity, the method also improved tests considerably. We tested business code in the non-distributed version. Such code itself is not trivial and the task would be considerably more difficult if we were already considering distribution issues, since distributed applications are considerably more complex than local ones. When following the steps of the method, we solely concentrated on distribution issues; during the test, we also focused on these issues. If, however, we came across a business error, then we would just turn off (simply by setting a variable in the configuration file) the distribution code, fix the business code, which could be done by a different developer, and turn on the distribution code again. This simplicity is achieved by flexible configuration provided by the factories mentioned in Step 4.

As pointed out before, distribution aspects were implemented simultaneously with persistent aspects, but after the GUI and the business layers so that the functionally complete prototype would be available as soon as possible, which is useful for validating user requirements. However, distribution issues could also be implemented at the same time than those layers, since the method relies on a modular architecture. This could be done, for instance, to assess risks related to distribution if they were critical to the application.

## 5 Related work

The work of Urban et al. [20], which was presented at the same time as Pim [4], uses source and target adapters in order to introduce CORBA-based distribution into an engineering application. As in this work, the aim is to isolate business from communication code. However, the adapters presented in that study perform extensive type transformation between CORBA and application types, which causes a 50% decrease in performance. The adapters used in our experiment do not perform such transformation, since the distribution platform is the same platform in which both client and server are implemented, namely, Java. Moreover, the pattern in that work does not use factories and is not explicitly supported by a method as in this work.

In [15] and [2], adapters are also used to isolate business code from RMI-based distribution. The first work deals with source adapters, and the second with target ones. The adapters in these guides are similar to the source and target adapters in this work, but there they act only as main service adapters, do not handle other non-functional requirements, such as caching and fault tolerance, and assume that the client has not been designed to acknowledge a general communication exception as the adapters in this work do. Additionally, those adapters are illustrated to serve only a single level of distribution, whereas the ones in this work may be applied to additional levels.

In [21], a downloadable proxy is used to handle additional non-functional requirements, such as selectively choosing when to make a remote call and when to make a local one. This is similar to the source adapter of an auxiliary service, but such proxy does not interact with a target adapter.

Aspect-Oriented Programming [9] could provide modular support for the introduction of distribution code. Such non-functional behavior can be treated as a unit, an *aspect*, whose specification is made separatedly and independently from business code. This aspect also has a similar role to the adapters in this work, and it affects business objects by transforming local calls into remote ones. The adapters and the other distribution code illustrated here could be implemented as aspects, and our method could be slightly adapted to support that.

## 6 Conclusion

The main contribution of this work is to propose a method for the implementation of distributed Java applications. This method guides the programmer in the non-trivial task of turning an initially centralized application into a distributed one. The method is both sound and useful.

Its soundness follows from implementing distribution issues modularly, thereby promoting software quality principles such as reuse and extensibility. In particular, the same business classes may be used in several applications and accessed by different distribution platforms, even in the same application. Additionally, changing business or communication code is easier for the programmer, since these are autonomous and, in fact, such task can be performed by different programmers, each one handling a different part.

We performed an experiment that has shown the method's usefulness. It is useful because it helps the programmer to tame the inherent complexity of implementing distributed applications. By following the method, the programmer will not tangle code of different layers, such as user interface, distribution, and business, thereby not having to deal with two different aspects at the same time. The method also improves tests, by making them simpler and more effective: once a problem in the business layer is detected, the communication code can be simply disconnected while such error is fixed; similarly, once a communication error occurs, business, user interface, and persistent code can be disconnected to fix such error.

The method refines Pim [4], but can also be used outside Pim to implement systems considering distribution in early development iterations. By being a part of Pim, the method in this work contributes to the improvement of current software development processes, which generally do not emphasize implementation issues [8], handling them in an *ad hoc* manner.

**Acknowledgments**

The authors wish to thank Tiago Massoni and Sérgio Soares for cooperation during the experiment. Augusto Sampaio also contributed with important suggestions to improve this work.

## References

[1] Vander Alves. Progressive development of distributed object-oriented applications. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Feb. 2001.

[2] Dan Becker. Design Networked Applications in RMI Using the Adapter Design Pattern. *Java World*, May 1999.

[3] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[4] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed Java applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, USA, 17th–18th May 1999.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[8] Ivar Jacobson et al. *The Unified Software Development Process.* Object Technology. Addison-Wesley, first edition, 1999.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP97– Object-Oriented Programming, 11th European Conference*, jun. 1997.

[10] Tiago Massoni. Um Processo de Software com Suporte para Implementação Progressiva. Master's thesis, Centro de Informática, UFPE, Feb. 2001.

[11] Object Management Group. *CORBA/IIOP Specification*, 2.3.1 edition, October 1999.

[12] Oracle Corporation. Oracle Databases, 2000. http://www.oracle.com/ip/deploy/-database.

[13] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern Oriented Software Architecture*, volume 2. John Wiley & Sons, 2000.

[14] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996. Pages 1–3.

[15] Gregg Sporar. Retrofit Existing Applications with RMI. *Java World*, January 2001.

[16] Sun Microsystems. *Java Remote Method Invocation Specification*, 1.50 edition, October 1998.

[17] Sun Microsystems. *Jini Architecture Specification*, 1.0 edition, January 1999.

[18] Sun Microsystems. Java Database Conectivity Specification, 2000. Available at ftp://ftp.javasoft.com/pub/jdbc.

[19] Sun Microsystems. Java Servlet Specification, Abril 2000. http://java.sun.com/-aboutJava/communityprocess/first/jsr053.

[20] Susan Urban, Ling Fu, Jami Shah, Ed Harter, Tom Bluhm, and Brett Hartman. The implementation and evaluation of the use of CORBA in an engineering design application. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 106–140, Los Angeles, USA, 17th–18th May 1999.

[21] M. Jeff Wilson. Get Smart with Proxies and RMI. *Java World*, November 2000.