Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications

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

1 Introduction

We introduce the Distributed Adapters Pattern (DAP) in the context of remote communication between two components, where it is intended that these components be decoupled from specific communication Application Programming Interfaces (API).

2 Context

In order to accomplish their tasks, components in a distributed system communicate with each other by means of an inter-process communication mechanism. When the components handle communication themselves we obtain applications where the core functionality of its components is interwoven with communication tasks. Therefore, the application becomes dependent on a particular communication mechanism, and its components are hard to reuse and extend.

In order to illustrate the use of DAP, we take a banking example as a concrete context. The banking service stores entities such as account and customer records, and has operations for manipulating these entities, such as `deposit` and `addAccount`. These operations are to be provided remotely to clients of the service, and thus its implementation must rely on a distribution platform. Additionally, it is expected that such implementation follows an incremental method: a non-distributed version is implemented before the distributed one. Another assumption is that it may be desirable to change the distribution platform.

3 Problem

Avoiding tangled communication and business code in order to provide reusability and extensibility.

---

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

4 Forces

DAP balances the following forces:

- Separation of concerns;
- The components should be independent from the communication API;
- The code modification in components to support communication should be minimized;
- Changing the communication mechanism should be a simple task, minimizing the impact on business code;
- Adequate communication performance;
- Development productivity must not be significantly affected.

5 Solution

Introduce a pair of object adapters [4] to achieve decoupling of components in distributed architectures. The adapters basically encapsulate the API that is necessary for allowing the distributed or remote access of Target objects (hereafter *Target object* refers to a business object providing services to other business objects). In this way, Source objects (hereafter *Source object* denotes a business object acting as a client of a Target object) of an application become autonomous with respect to the distribution layer, so that changes in the latter do not impact the former.

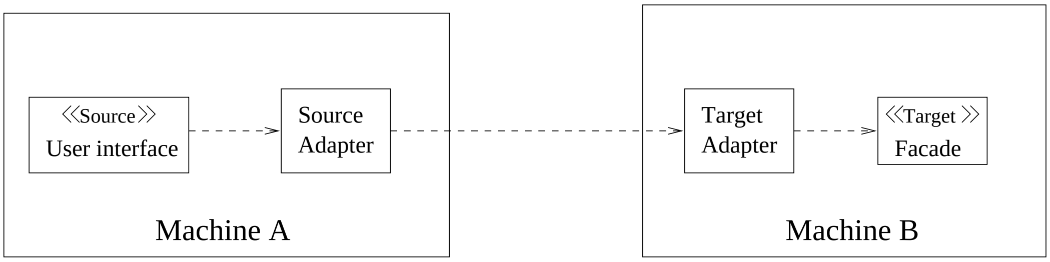


Figure 1: An example of DAP.

There are two kinds of adapters: *source adapters* and *target adapters*. Roughly, the latter wraps Target 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 [4] object co-located with the target adapter. Figure 1 illustrates this example.

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.

As another example, the banking application of Section 2 is structured according to DAP as Figure 2 illustrates.

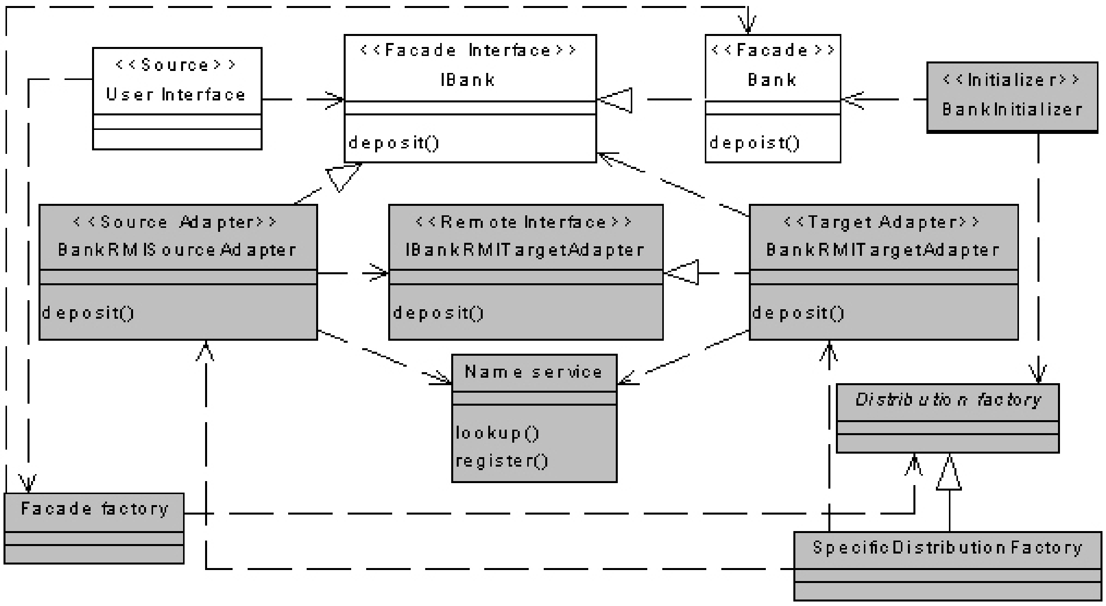


Figure 2: Class diagram of a banking application according to DAP.

The uncolored elements deal with the business aspects of the application, whose Facade is the `Bank` class, which unifies all services of the application. The gray elements denote the adapters and their collaborators. Essentially, these gray elements hide distribution API from user interface and business code. In the following section, each element is described abstractly; their implementation is sketched in Section 8.

5.1 Structure

Figure 3 details the structure of DAP by means of a class diagram. The `Source` and `Facade` classes abstract business components as mentioned previously. The `Facade` class is named after the Facade design pattern [4]. The `Facade Interface` abstracts the behavior of the `Facade` class in a distributed scenario. However, this interface, the `Source` and `Facade` classes have no communication code. These three elements constitute a distribution-independent layer in the pattern. The remaining elements of the pattern deal with this aspect.

The core elements of the pattern handling distribution itself are `Source Adapter` and `Target Adapter`. These are tied to a specific distribution API and encapsulate the communication details. `Source Adapter` is an adapter [4], isolating the `Source` class

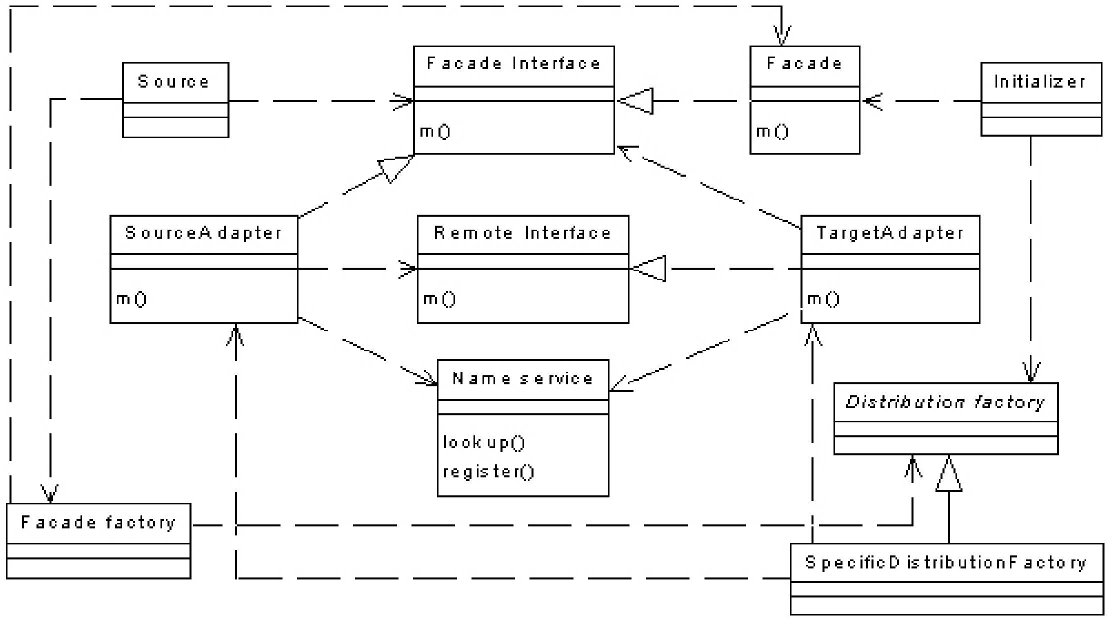


Figure 3: Class diagram of DAP.

from distribution code. It resides on the same machine as the `Source` and also works as proxy [4] to `Target Adapter`. This latter may reside on another machine and is also an adapter, isolating the `Facade` class from distribution code. Since `Source Adapter` and `Target Adapter` usually reside in different machines, and thus do not interact directly, `Target Adapter` implements `Remote Interface`, on which `Source Adapter` depends.

The `Name Service` class has operations for registering and looking up a remote object; both adapters use this class, which represents a generic name service and is common to most distribution platforms. The `Initializer` class also resides in the same machine as `Target Adapter` and `Facade`, and is responsible for creating `Facade` and `Target Adapter` objects. Its importance lies in the fact that it allows the same `Facade` object to be accessed at the same time by different target adapters, representing different distribution technologies. Concurrency control is orthogonal to distribution and can be studied in books such as [5]. The factories in the pattern are useful for configuration purposes: they are used in the creation of `Facade` and of the adapters. In particular, the factories isolate business code from the creation of adapters for a specific distribution platform.

5.2 Dynamics

Figure 4 shows the sequence diagram of a typical scenario for DAP. The `Initializer` creates a `Facade` object and a `Target Adapter`[1], passing to the latter a reference to the former. `Target Adapter` registers itself as a distributed object in the `Name Service` by

[1] Actually, `Initializer` delegates the creation of this adapter to `DistributionFactory`. We omit it here for simplicity.

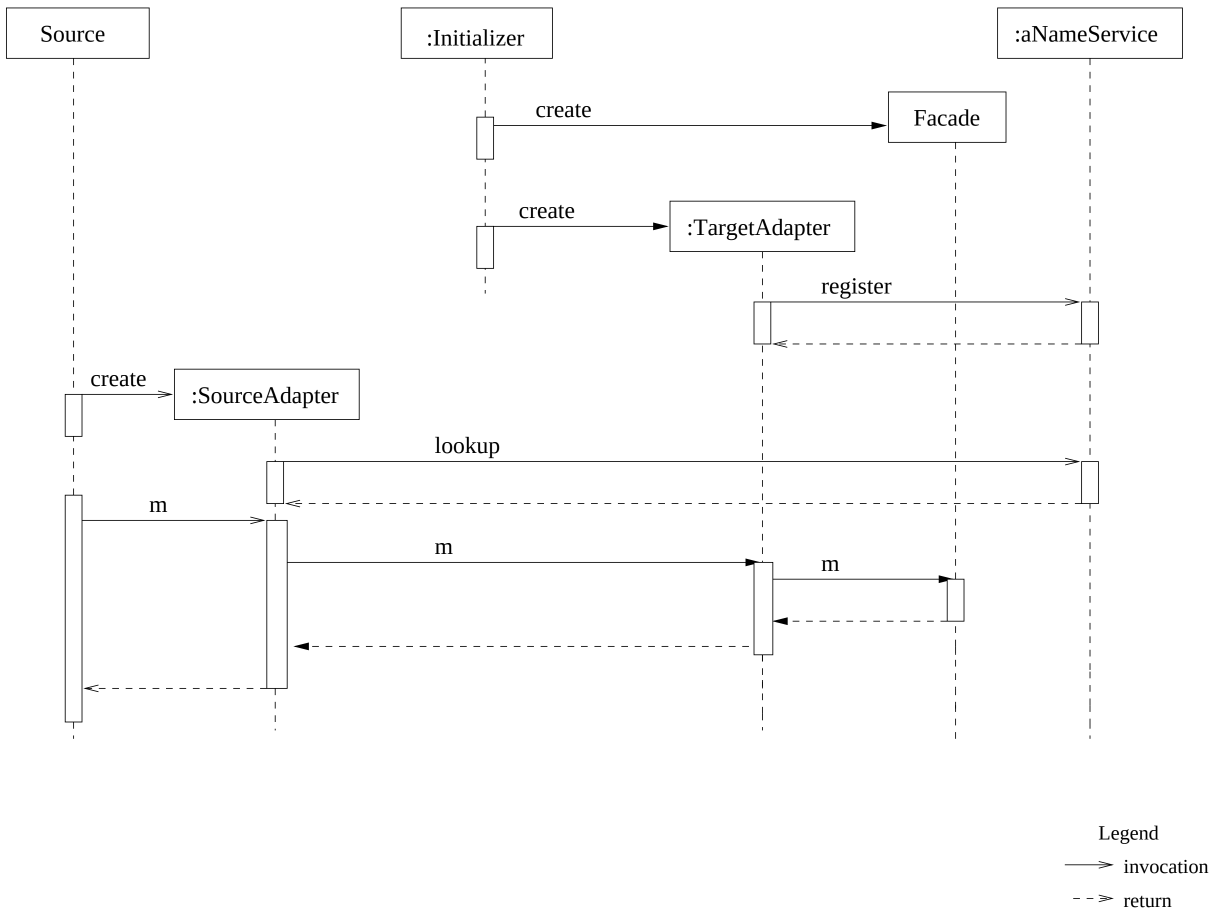


Figure 4: Dynamics of DAP.

invoking its `register` method. During initialization, `Source` creates a `Source Adapter`[2], which performs a `lookup` operation on `Name Service` to obtain a reference to the remote service offered by `Target Adapter`. `Source` then invokes the local `m` operation on `Source Adapter`, which in turn calls the remote `m` operation of `Target Adapter`; this latter delegates the call locally to `Facade`.

6 Consequences

DAP provides the following benefits:

- *Modularity.* This pattern separates concerns by structuring 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 the Source and Target components easily in other applications based on other APIs and middleware technologies. In addition, changes to the middleware aspects are simpler, since these are restricted to the distribution layer.

[2]In fact, `Source` delegates this to `FacadeFactory`.

- *Incremental implementation.* The pattern supports incremental implementation. During the early phases in development, developers construct a functionally complete prototype, where the Source component (a GUI, for example) depends directly on the Target component (a business Facade, for example). Later, developers add the distribution layer seamlessly, since this latter implements the same interface as the Target component.

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

7 Implementation

For example, here we consider how to implement the Distributed Adapters Pattern using RMI [9] as the distribution technology. Consider the following implementation issues:

- *Serialization of business objects.* As RMI supports a value parameter passing mechanism for local objects, the classes of these objects must implement the `Serializable` interface [9]. There are no methods in this interface and it simply indicates to the RMI system that an object may be transformed into a stream of bytes in order to be transmitted over a network. However, this is not a negative dependence between the business and the distribution layers since the former calls no method on the latter; in fact, no change on the latter will affect the former.

- *Additional non-functional requirements.* RMI is a simple distribution platform and does not offer fault-tolerance and caching. Such extended behavior can be implemented in DAP's adapters (a detailed implementation is presented in [1]).

8 Sample Code

We now provide sample code for the core elements in the pattern, using the simple banking application mentioned in Section 2 as an example (a full implementation is given in [1]). This application is structured according to DAP as shown by Figure 2. The `Bank` class is a `Facade`, and it keeps references to entities such as account and customer records, and has operations for manipulating these entities:

```
class Bank implements IBank {
   private AccountRecord accounts;
   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 to or withdrawing from them. The exception `UnknownAccountException` is specific to the banking application. The `IBank` interface implemented by the banking facade is a `Facade Interface`. It abstracts the behavior of the application:

```
interface IBank {
   void deposit(String accountNumber, double value)
          throws CommunicationException,
                 UnknownAccountException;...
}
```

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

A `User interface` object simply creates a `BankRMISourceAdapter` and forwards client requests to it. The RMI source adapter implements `IBank` so that the `User interface` class is unaware of the specific middleware technology. The constructor obtains a reference to the target adapter, by invoking the `connect` method:

```
public class BankRMISourceAdapter implements IBank {
   private IBankRMITargetAdapter  bank;
   public BankRMISourceAdapter(String whereServer)
            throws CommunicationException {
        connect(whereServer);
   }
   public void connect(String server) throws CommunicationException {
      try {
           bank = (IBankRMITargetAdapter) Naming.lookup(server);
      } catch (Exception e) {
          throw new CommunicationException (...);
        }
   }
```

A `User interface` object can call the `connect` method later in case the connection with the target adapter fails (in fact, the source adapter itself may implement fault-tolerant behavior as described in [1]). The `deposit` method forwards `User interface` deposit requests to the target adapter:

```
   public void deposit (String accountNumber, double value)
           throws CommunicationException,
                  UnknownAccountException {
      try {
          bank.deposit(accountNumber,value);
```

```
      } catch (RemoteException e) {
           throw new CommunicationException (...);
        }
    }
} //end of BankRMISourceAdapter
```

Note that, both in the constructor and in the `deposit` method, the source adapter replaces an RMI specific exception with the general `CommunicationException`. The `IBankRMITargetAdapter` interface is the type of the reference to the target adapter and its methods must also raise `RemoteException`:

```
public interface IBankRMITargetAdapter extends Remote {
   void deposit(String accountNumber, double value)
      throws CommunicationException, UnknownAccountException,
             RemoteException;
}
```

where `Remote` is an RMI interface used to identify remote object types [9].

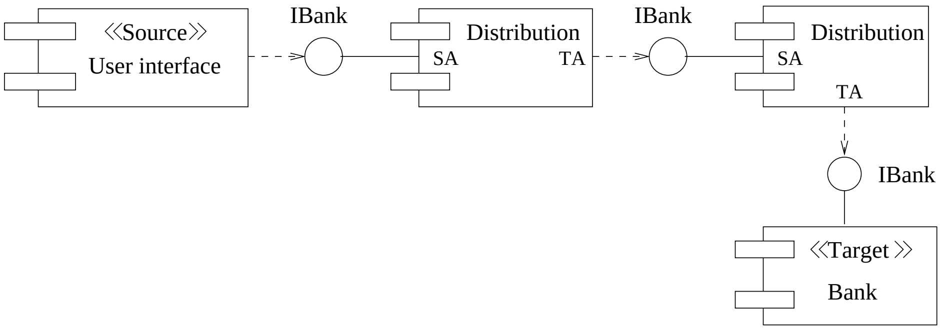
The target adapter becomes an RMI remote object by inheriting from the `UnicastRemoteObject` [9]. It implements the `IBankRMITargetAdapter` remote interface, so that the source adapter can call its methods remotely. The constructor of the target adapter receives a facade object as an argument and registers the adapter itself in the name service:

```
public class BankRMITargetAdapter extends UnicastRemoteObject
          implements IBankRMITargetAdapter {
   private IBank bank;
   public BankRMITargetAdapter(IBank bank)
              throws  CommunicationException {
      try {
           this.bank = bank;
          Naming.rebind("BankServer", this);
      } catch (Exception e){ throw new CommunicationException(...);}
   }
```

The source adapter invokes the `deposit` method on the target adapter, and this operation forwards the call to the corresponding method in the facade object:

```
   public void deposit(String accountNumber, double value)
            throws CommunicationException, RemoteException,
                   UnknownAccountException {
      bank.deposit(accountNumber, value);
   }
} // end of BankRMITargetAdapter
```

Note that the type of the target adapter's attribute is `IBank` and not `Bank`. The rationale is that, since either a facade or a source adapter implements `IBank`, the target adapter, which depends on this interface, may refer either to a facade or to another



This figure illustrates an application with two levels of distribution. Each distribution component abstracts both adapters. SA and TA denote Source Adapter and Target Adapter, respectively.

Figure 5: Additional levels of distribution.

source adapter. This latter case accounts for flexible configurations where there are additional levels of distribution. Figure 5 illustrates this situation. As mentioned previously, the methods of the `IBank` business facade interface declare `CommunicationException`. Therefore, methods in the target adapter and in its remote interface must also declare this exception.

9 Known Uses

DAP has been used in the implementation of a Web based information system, where the adapters are used between the web server, in which servlets [10] act as clients of the source adapter, and the application server, in which the target adapter interacts with the facade. The facade is not in the web server due to security and performance reasons.

Another use of DAP in Web based information systems employs the adapters between an applet, in a client Web browser, and a facade, in a remote machine. The adapters hide the communication details, which use HTTP [12], from the applet and the facade.

The work in [8] and [2] reveals that developers have been using patterns that have some relation to DAP. In particular, the pattern in the first work is similar to DAP's source adapter; the pattern in the second work is similar to the DAP's target adapter.

10 Related Patterns

- *Distributed Proxy Pattern [6].* This pattern and DAP have similar objectives. However, following to [11], DAP does not attempt to make the incorporation of distribution totally transparent. Indeed, a client of a source adapter in DAP must be prepared to handle the general `CommunicationException`. DAP makes transparent the use of a particular distribution technology, not distribution itself. In order to achieve it, DAP uses adapters (instead of proxies), which replace specific distribution code by general code, for example by turning `java.rmi.RemoteException` into

`CommunicationException`. Moreover, the adapters in DAP may implement additional non-functional requirements, such as fault-tolerance and caching, and may also be used to achieve `n` levels of distribution (as shown in Figure 5), each of which may be implemented by a different technology.

- *Wrapper-Facade [7]* and DAP have the common goal of minimizing platform-specific variation in application code. However, Wrapper-Facade encapsulates existing lower-level non-object-oriented APIs (such as operating systems mutex, sockets, and threads), whereas DAP encapsulates object-oriented distribution APIs, such as RMI and CORBA.

- *Adapter, Facade, and Abstract Factory.* DAP is implemented using the Adapter, the Facade, and the Abstract Factory design patterns [4].

- *Broker and Trader.* Well known patterns for structuring distributed systems already exist. The Broker [3] and Trader [3] patterns are examples. These are architectural patterns 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. DAP uses these fundamental patterns and provides a higher level of abstraction: distribution API transparency to both clients and servers.

- *Chain of Responsibility [4]* is similar to DAP in the sense that it decouples the sender of a request from its receiver by giving more than one object the chance to handle the request. This indirection is similar to the DAP's adapters; these, however, also perform interface filtering, isolating the distribution platform's API, which is not done by Chain of Responsability.

- *Model-View-Controller (MVC) [3]* is used in the context of interactive applications with a flexible human-computer interface. Its goal is to make changes to user interface easy, and even possible at run time. DAP is used in the context of distributed applications and aims at making changes to the distribution platform a simple task.

Acknowledgements

We would like to thank our shepherd, Eduardo Fernández, for all the work he put into commenting on this paper and the great suggestions for improvement he made. During the writer's workshop at SugarLoafPLOP'2001, Jorge Ortega Arjona, Gunter Mussbacher and Sérgio Soares have also made several interesting comments that helped to improve this paper.

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

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

[5] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999.

[6] Antonio RitoSilva, Francisco Rosa, and Teresa Goncalves. Distributed proxy: A design pattern for distributed object communication. In *PLoP'97*, Monticello, USA, September 1997. http://jerry.cs.uiuc.edu/~plop/plop97/Proceedings/ritosilva.pdf.

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

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

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

[10] Sun Microsystems. Java Servlet Specification, Abril 2000.

[11] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems, November 1994.

[12] The World Wide Web Consortium. *Hypertext Transfer Protocol Specification*, 1.1 edition, jun. 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.