

Casting an Abstract Design into the Framework of Java RMI

Klaus Bergner, Andreas Rausch, Marc Sihling

Institut für Informatik
Technische Universität München
D-80290 München, Germany
<http://www4.informatik.tu-muenchen.de>

Abstract

Java's Remote Method Invocation mechanism is an object-oriented middleware framework for the development of distributed applications. We describe our experiences with a case study where we casted an abstract design specification for a small, distributed multi-user editor into the framework of Java RMI. We concentrate on the encountered difficulties and present workarounds for certain problems.

1 Introduction

The Java language framework was first presented by SUN in 1995 and has since been continuously developed further. With version 1.1 of the Java Development Kit [12], various enhancements have been introduced, especially in the area of the graphical user interface. Other new features are *Object Serialization* and an object-oriented remote procedure call facility named *Remote Method Invocation*, or just RMI [13].

Object serialization offers a mechanism to store an object together with all of its referenced objects to a stream of bytes and to safely restore the objects from the byte-stream later. By mapping the stream to a file it is very easy to store object graphs persistently.

RMI allows the communication between objects in different processes and address spaces, possibly on different hosts. As soon as a Java program gets a reference to a remote object—either via parameter passing or via a special bootstrap-naming service—it can send method calls to this object in a transparent way. The RMI mechanism takes care of marshaling and unmarshaling parameter objects using object serialization.

RMI and object serialization are tightly integrated into the Java framework and extend Java features like garbage collection and dynamic binding to support distributed programming. Together, these two techniques form a small object-oriented middleware framework for the development of distributed Java applications.

It is an interesting question how such a framework influences the development of a system: Can an abstract design be casted seamlessly into the framework? Does one have to adjust or even to restructure the design in order to use RMI? How easy is the translation from existing, non-distributed Java programs?

We have examined these questions in the context of a case study targeted primarily to the evaluation of the UML modeling language [3]. This paper contains only the part of the case study results concerned with the design specification and its translation to an RMI-based implementation. Most of the UML diagrams and our comments and evaluations concerning the description techniques and their interrelationships are also left out—they can be found in the complete case study documentation [1].

The rest of the paper is organized as follows: Section 2 contains information about the project of the case study and the process we followed. Section 3 describes the abstract design and how it was mapped to a distributed client/server architecture, and Section 4 concerns its realization using RMI. A short conclusion ends the paper.

2 Project and Process

2.1 The Project – A Distributed Schedule Planner

The project of the case study was to develop a small, distributed editor for planning break supervision schedules in

big schools. According to the customer specification [11] which was prepared by the DACH group [6] the application scenario is as follows:

Teachers have to supervise pupils in the various parts of a school building during the breaks. The assignment of teachers to breaks is specified in the break plan of the respective building part. Each break must be supervised by a teacher, and teachers are assigned to breaks depending on the time they spend for teaching—a full-time teacher has to supervise more breaks than a teacher with only a couple of lessons per week. Teachers can provide the school with exclusion time periods during which they can not be assigned to breaks because of other duties.

The intended system supports the persons responsible for maintaining the break plans and the teaching staff data. The tool allows the users to create, to edit, and to delete break plans, to assign teachers to breaks, and to manage a list of the school's teachers. The data of a single break plan—and also the data of the teaching staff—can be edited by at most one person at the same time, although users may work in parallel on different break plans at the same time.

Additionally, the tool has to maintain global statistics, containing, for example, the number of breaks a teacher still needs to be assigned to. The tool must also immediately notify the users whenever the current break assignment configuration has a conflict, which may happen because of the following reasons:

Exclusion Overlap: A teacher is assigned to a break overlapping with one of his or her exclusion times.

Break Conflict: A teacher is assigned to two concurrent breaks on different break plans.

The target hardware is the school's local area network, consisting of some PCs with standard, Java-capable web browsers. The break planner system has to be implemented using Java applets in order to ease the deployment of the software and to enable the users to work at home over a dialup-connection.

2.2 The Process

For reasons of clarity, we have chosen to structure the development documentation according to the phases of a typical waterfall model. Our actual process was not so linear because of some feedback loops between the phases. In the context of this paper, only the design phase is interesting.

System design is concerned with the development of an abstract technical solution independent of a certain implementation language or framework. We splitted this phase fur-

ther into two sub-phases, following the principle "Architecture first—distribute later." (cf. [2]). During *business-oriented design* (see Section 3.1), design classes are added to the classes found during system analysis, and the decisions about the operations and attributes, the intended object graphs at runtime, and the flow of control and data are made. During *distribution design* (see Section 3.2), the distribution of the objects on physical computation nodes and the communication protocols to be used are determined. Finally, the resulting design was implemented using Java RMI (see Section 4).

3 System Design

Our strategy during this phase was to design the business-oriented data and functionality of the system before determining its distribution architecture. Apart from providing additional structure for the development documents, this has the advantage that many basic design decisions can be met without getting involved with the complexities of the underlying distribution architecture. Because functional and non-functional aspects are clearly separated, the functional design is more or less independent of the technical aspects of a certain distribution architecture, simplifying the transition to other distribution architectures.

3.1 Business-Oriented Design

The essential task during business-oriented design was to construct a detailed class diagram. Setting the design focus on the classes of the system makes the transition to the final implementation easy because classes are the prevalent structuring construct of object-oriented program code.

Figure 1 shows the business-oriented design class diagram. The contained classes and associations are explained in the following subsections (Sections 3.1.1 to 3.1.4). The attribute and method names of the classes are mostly self-describing. Because of the lack of space we have neither included a data dictionary nor graphical description techniques for them like in [1]. However, some of the more complex methods are explained informally.

3.1.1 Application Classes

Most of the classes come from our analysis class diagram. It is not contained in this paper, but can be found in [1]. **Staff** represents the teaching staff of the school. It has the stereotype «Singleton» to specify that only one instance of this class can exist at runtime. The teaching staff consists of teachers, represented by **Teacher** objects that are associated with **Break** objects via the **supervises**-association and

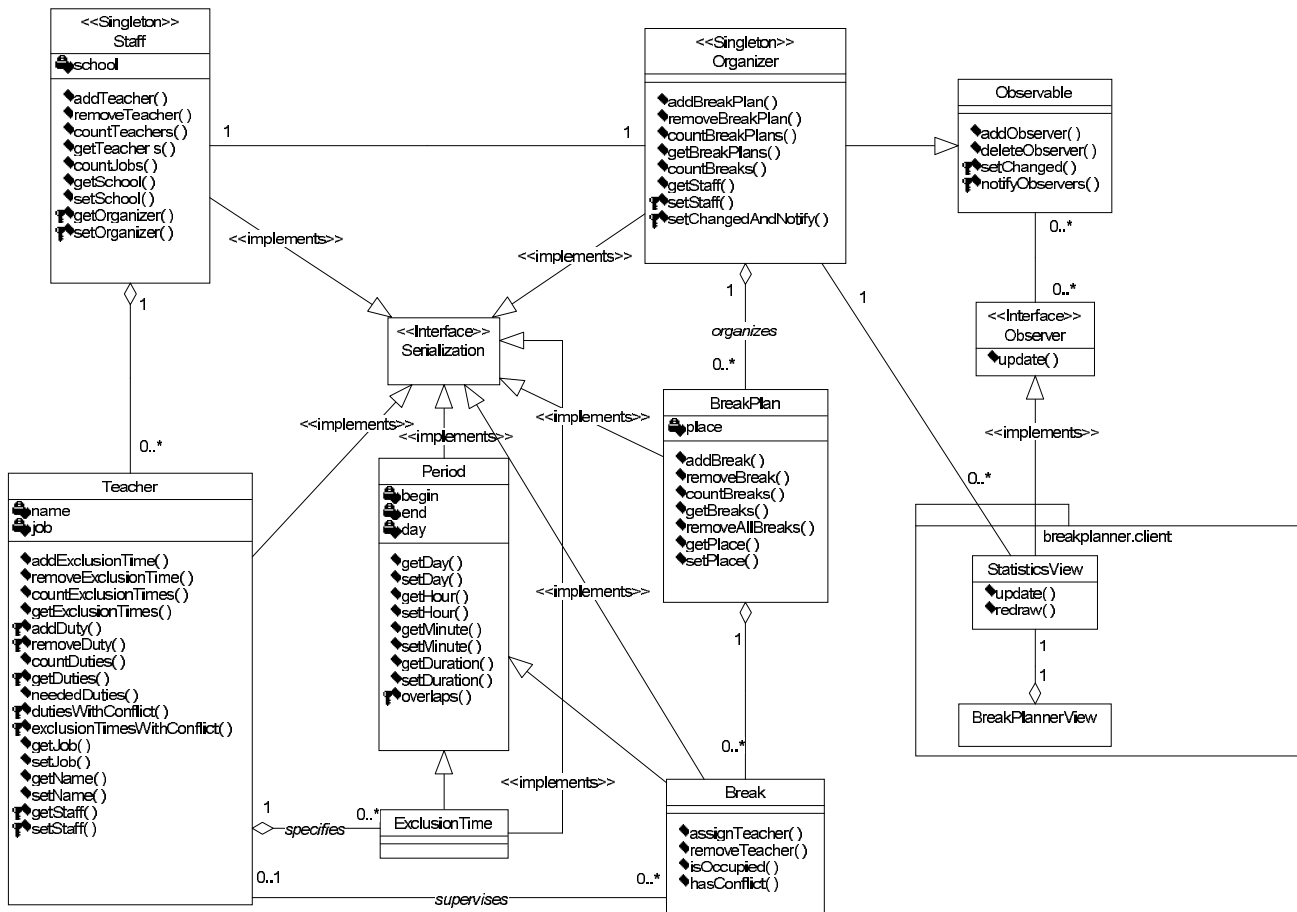


Figure 1. Business-Oriented Design Class Diagram

to ExclusionTime objects via the specifies-aggregation. The base class Period captures the common characteristics of Break and ExclusionTime. It represents a weekly recurring period of time, like “each Monday, from 10:00 to 12:00”. Each Break object is part of a BreakPlan object, representing a break plan for a special building part to be supervised by a teacher. Finally, Organizer stands for the collection of all break plans of a school. Its singleton object thus serves as the root of the object hierarchy at runtime.

3.1.2 User Interface

The package breakplanner.client contains all view classes for the break planner application’s GUI. The view classes represent the visual interface elements of the GUI responsible for the presentation and manipulation of the data.

Although there exist many different view classes for the presentation of the different data entities of the break planner, we have modeled only two exemplary classes. Statis-

ticsView controls a window with the statistics data, and BreakPlannerView represents the main window of the break planner application. We think that the decision to leave out most of the view classes is reasonable because these view classes can be “modeled” and implemented easily with the help of an interactive GUI tool. Yet, we wanted to include at least one of the view classes into our design because it is needed to model the interaction between the application’s GUI and the system core (for a detailed explanation see the next section).

3.1.3 Realization of Update on Change

An advanced requirement for the break planner application is the immediate update of the user interfaces: Each time a user changes a data value, the break statistics has to be updated, and possible conflicts have to be visualized to all users. To realize this *Update on Change* policy, we have used the so-called “observer pattern” (see [7] for de-

tails). The implementation of the observer pattern in Java is done via implementation/extension of the available standard Java classes/interfaces `Observer` and `Observable`. To distinguish Java interfaces from ordinary classes, they have been given the stereotype `«Interface»`. Analogously, implementation relations are marked with the stereotype `«implements»` to distinguish them from ordinary inheritance relations.

The collaboration diagram of Figure 2 shows the dynamic behavior of the observer pattern: `Observer` objects register at the `Observable` objects they are interested in by calling the latter's `addObserver()` method; unregistering is done via calling `deleteObserver()`. Each time a user changes an `Observable`'s data, the `Observable` calls the `update()` method of each registered `Observer`. The called `Observer` can then react on the change of the `Observable`, for example by requesting the modified data from the `Observable`.

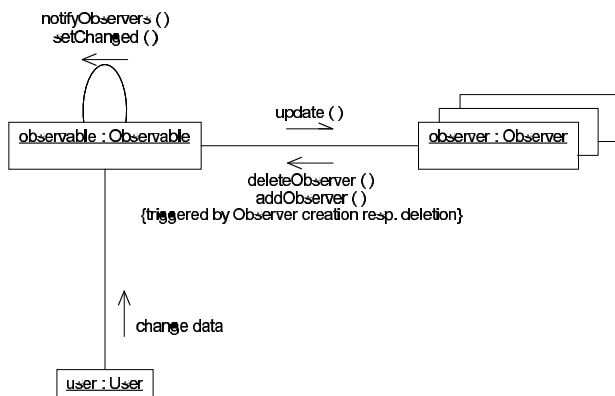


Figure 2. Collaboration Diagram of the Observer Pattern

3.1.4 Persistence Management

The break planner application needs to store its data persistently because the information about breaks plans and staffs is valid for long periods of time and must survive multiple runs of the system. For our system we decided to use the standard Java object serialization mechanism in combination with plain files because this seemed sufficient for the management of the relatively small amount of data. Furthermore, this mechanism can be easily used by simply deriving classes from the interface `Serializable`. In the class diagram of Figure 1, this was done for all application classes. Object graphs consisting of objects of these classes can then, for example, be provided to a standard Java `ObjectOutputStream` and mapped to a file.

Note that `Break` and `ExclusionTime` implement `Serializa-`

tion, although the base class `Period` already does so. The reason for this is that serializability is not transitively inherited in Java, but can be given each class independently of its base classes.

The choice of a persistence mechanism usually constrains the distribution architecture. The decision to use a monolithic object-oriented database system would, for example, in most cases lead to an architecture where the data is centralized on the database server. This holds also for our system: Although object serialization is a Java feature that can be used everywhere—on a client as well as on a server—, one can not store data persistently from within a client applet running on a Java-capable browser (cf. Section 3.2.1).

3.2 Distribution Design

Distribution design is concerned with the partitioning of the data and functionality of a system on a network of physically or logically distributed computation nodes. At this point, the constraints induced by the target hardware and the base software system have to be considered.

3.2.1 Implications of Target System

The target system is the school's local area network, consisting of some PCs with standard web browsers. As said in Section 2.1, the break planner system has to be implemented using Java applets in order to ease the deployment of the software and to enable the users to work at home over a dialup-connection.

One of the requirements stated during requirements analysis was that the target system consists of a local area PC-network and has to be implemented using Java applets (see Section 2.1). The distribution architecture is restricted considerably by this requirement because it implies that GUI objects are managed by applets running on client computers.

The requirement also implies the existence of at least one server with a "real" Java application that can handle persistent information—applets running on browsers are usually forbidden to access local files according to the sandbox safety model of Java [8].

3.2.2 Distributing the Application Objects

Distributing the break planner's application objects is more difficult. A first approach is suggested by Section 2.1. It states that a single break plan—and also the teaching staff—can be edited by at most one user at a time. This seems to imply a simple check-out/check-in solution for break plans,

where users check out break plans from a central repository, edit them locally, and check them in again. Such an architecture has the advantage that interactive editing is very fast because it is performed locally without communication overhead between distributed nodes.

However, a closer inspection shows that a simple check-out/check-in architecture with purely local editing is not a proper solution because of the requirements of the global break statistics and conflict detection. If a user assigns a teacher to a break in a single break plan, the statistics views of all other users have to be updated, and possible conflicts must be visualized to them. To fulfill these requirements, we have considered two alternatives:

Enhancing the Check-Out/Check-In Solution

One possibility is to send change notification messages from each client to all other clients on each break plan update. This could be implemented easily if Java provided a transparent object migration facility keeping track of references to mobile objects. However, because such a mechanism does not exist in RMI, it would require the implementation of a proprietary, albeit small object request broker doing all the book-keeping. We can also imagine a variant with replication where all break plans are duplicated on the server, and the changes on clients are written through to the server so that the other clients can observe them.

Holding All Application Objects on the Server

The other possibility is to hold all application objects on the server and to put only the view objects on the clients.

The essential advantage of the second alternative is that it is simple and robust and leads to a flexible, easily extendable design. As all application state is kept on the server, one does not have to deal with any of the consistency problems of a truly distributed solution. The only drawback is that interactive editing of break plans is slower than with the first alternative because the clients have to access remote server data for each user action. However, we believe that the delays will be tolerable in a small school network with low network traffic, and chose the second alternative.

3.2.3 Restricting the Client's Interface

On the one hand, a client of a class should be offered sufficient functionality to use the class effectively and comfortably for its intended purpose. On the other hand, clients should not be allowed to access any additional functionality. This very important design principle makes it easy to change the implementation of methods hidden from the

client—it is also known as the principle of “shallow interfaces” or “loose coupling”.

Having a look at our example we can distinguish two different users of the implementation classes:

- Clients have access to a rather limited interface. Apart from conflicting with the principle of shallow interfaces, granting all clients access to all server features would open a potential security hole. Clients are, for example, not allowed to connect a `Teacher` to a `Break` via the method `Teacher::addDuty()` because this method does not ensure the bidirectionality of the `supervises-association`, as `Break::assignTeacher()` does.
- The server must have access to the full functionality of the implementation classes.

To realize the principle of shallow interfaces, we have distinguished client-visible functionality from server-only functionality (the technical details are explained in Section 4.3).

3.2.4 Introduction of Server Functionality

In order to be accessible for a client, remote server objects must first be activated. In our design, the newly introduced class `BreakPlanner` takes care about the activation/deactivation of the remote objects on the server by initiating the loading/storing of persistent application data whenever it is created/destroyed (see also the complete class diagram of Figure 3 which is explained in detail in the following section).

Furthermore, clients must be able to get remote references to server objects. This is usually done by providing a server-side naming service that binds names to certain “hook” objects and exports these names to the clients. We use the `Organizer` server object for this purpose: At runtime, it is registered via the `rmiregistry` mechanism [13]. This allows clients to access the entire application object graph, following, for example, the reference from the `Organizer` object to the `Staff` object.

4 Realization with RMI

4.1 Client Interfaces and Server Implementation Classes

The changes between the business-oriented and the distribution-oriented architecture class diagram (see Figures 1 and

3) are simple and almost schematic, as proposed in SUN's tutorial for Java RMI [13]: Each class whose objects must be accessed remotely is split up into an interface and an implementation class. The interfaces contain the functionality used by the client; they are derived from the standard interface `Remote` and are given the names of the original classes. The implementation classes are used on the server; they are derived from class `UnicastRemoteObject`, implement their corresponding remote interfaces, and their names are suffixed with `Impl`. An example is the business-oriented design class `Break` which was split up into the distribution-oriented design class `BreakImpl` and the corresponding interface `Break`.

4.2 Remote Observer Mechanism

Instead of the standard Java class `Observable`, a remote version has to be used because observer and observable objects are on different sides of the client/server-gap. During the implementation phase we found ourselves unable to combine Java's observer mechanism with the RMI concept. Neither inheritance nor wrapper-based techniques make it possible to create a remote observer mechanism on top of Java's standard observer classes. The only remaining solution is a total re-implementation of the needed functionality. Fortunately, we did not have to do this by ourselves because we could use a free implementation provided by Caribou Lake Software [5]. Note that the name of the observable implementation class of this package violates the usual naming conventions: Instead of `COM.cariboulake.util.Observable` it should better be `COM.cariboulake.util.RemoteObservableImpl`.

Note also that the GUI class `StatisticsView` has to be a remote class, too, because its objects observe the `Organizer` object lying on the application server. Therefore, these client objects must themselves be remote servers for the `Organizer` object's callbacks to their `update`-method. It is interesting that this requirement restricts the target platform quite seriously: It disallows access to the server from outside a firewall because SUN's current RMI implementation uses HTTP tunneling in this case, and HTTP usually does not allow bidirectional control flow [13].

4.3 Realization of Client Interface Restriction

The restriction of the client's functionality can be seen in the class diagram of Figure 3: Most client interfaces contain only parts of the functionality of their corresponding server implementation classes. The `PeriodImpl` class has no client interface at all and is, therefore, hidden from the client entirely.

A problem with this approach is that the server needs to cast objects from interface types to server types in order to use the additional server functionality, as can be seen from the following example: Imagine that the client wants to assign a teacher to a break by calling `Break::assignTeacher(t)` where the parameter `t` is of type `Teacher`. If `Break::assignTeacher(t)` now wants to call `t.addDuty()`, it must first cast `t` to type `TeacherImpl`.

Unfortunately, such casts are not easily possible in the framework of Java RMI. If a client sends the server a reference to one of the server's own remote objects, the server can not typecast that reference to the corresponding implementation object. The reason for that is that the server does not get a local reference, but instead a reference to a so-called "stub" object that accesses the "real" implementation object via a "skeleton" (see the RMI documentation [13] for details). To invoke server-only methods—methods not contained in one of the remote interfaces accessible to the clients—, one has to maintain a table with this relation on the server. In our implementation [10], we use the extra class `InterfImplHandler` for that purpose.

To clearly separate the server and client functionality, we have provided different, but semantically equal methods for each operation accessible for the client as well as for the server:

Server Methods are declared only in the server implementation classes and implement the functionality of the method. Their signatures contain only implementation class types (the ones suffixed by "Impl"). To distinguish these methods from the client methods, their names are prefixed with the letter 'i' (for implementation).

Client Methods are declared in the client interface and implemented in the server implementation class. They provide no own functionality, but serve only as a wrapper for the server methods. Their signatures contain only interface types. Client methods have a very simple, schematic implementation consisting of the following actions:

1. Find the corresponding implementation object for the stub parameter object via the `InterfImplHandler` object.
2. Cast down the types of all actual parameters from remote interface types to server implementation types. This step can be done implicitly as part of the first step by storing references of server implementation types instead of remote interface types in the `InterfImplHandler` table.
3. Call the corresponding server method.

4. Cast up the type of the server method result parameter—provided one exists—to a remote interface type and return it to the caller.

Apart from the gained clarity of the implementation, this scheme has the advantage that the server functionality can be easily tested stand-alone by considering only server methods. If the server is stable enough, one can then deal with the additional issues introduced by distribution. Another advantage is the improved performance. Once the translation to server objects has been done by the client method, all its method calls to the parameter objects on the server run locally and do not involve stubs or remote references.

5 Conclusion

We have cast the complete business-oriented design into the framework of RMI. The interesting and essential point is that this transformation from a universal, abstract design to a concrete technical framework could be done almost schematically once the basic client/server distribution architecture has been chosen.

All in all, our valuation of Java RMI is quite positive—the mechanism is easy to learn and can be used more or less transparently. Our main complaints are:

- RMI objects are made remote by inheriting from class `RemoteObject`. This forces the programmer to make all classes in a hierarchy remote if actually only one class is intended to be remote. Although this was no problem for our application, “disinheriting” non-remote classes from remote ones during distribution design could necessitate extensive transformations of the class hierarchy and the corresponding algorithms.
- It can be difficult to make existing Java classes remote. An example is the remote observer mechanism that can not be implemented by reusing Java’s existing observer classes.
- One can not easily restrict a client’s functionality without restricting the server’s functionality, too. However, a schematic workaround can be applied that uses client methods to transfer remote references to local server object references, thus uncovering the restricted functionality on the server.

It is an interesting question whether and how RMI will survive and evolve in the future. To examine the advantages and disadvantages with respect to its main contenders, CORBA [9] and DCOM [4], we plan to redo our case study using these techniques.

Acknowledgements

This paper originated in the subproject A1 on “Component-Based Software Engineering” of the ForSoft research cooperation (“Bayrischer Forschungsverbund Software Engineering”) and was supported by Siemens ZT.

References

- [1] K. Bergner, A. Rausch, and M. Sihling. Using UML for modeling a distributed Java application. Technical Report TUM-19735, Technische Universität München, Institut für Informatik, 1997.
- [2] S. C. Bilow and P. S. Bilow. Distributed systems design. In *OOPSLA’95, Addendum to the Proceedings*. ACM Order Department, 1995.
- [3] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language, Version 1.0*. Rational Software Corporation, URL: <http://www.rational.com>, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951 (USA), 1997.
- [4] K. Brockschmidt. *Inside OLE2*. Microsoft Press, 2nd edition, 1995.
- [5] Caribou Lake Software. *Remote Observer Classes*, <http://www.cariboulake.com/utills.html>, 1997.
- [6] DACH Group. Universität Hamburg, FB Informatik, AB Softwaretechnik; Johannes-Kepler-Universität, Linz, Austria, Institut für Wirtschaftsinformatik, Doppler-Labor für Software Engineering; GMD Bonn, Schloß Birlinghoven, St. Augustin; UBS Information Technology Laboratory, Zürich, Switzerland.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [8] Java security – frequently asked questions, 1997.
- [9] Object Management Group. OMG website, <http://www.omg.org>.
- [10] A. Rausch and M. Sihling. Source code for the break planner application backend, <http://www12.informatik.tu-muenchen.de/public/seep98/bp.tar.gz>, 1997.
- [11] S. Roock, K.-H. Sylla, C. Lilienthal, and A. Müller-Lohmann. Der Pausenplaner – Szenario, CRC-Karten, Systemvision, <http://set.gmd.de/~sylla/dachpap-aufgabe.html>, 1996.
- [12] SUN Microsystems. *The JDK 1.1.2 Documentation*, <http://java.sun.com/products/jdk/1.1/docs>, 1997.
- [13] SUN Microsystems. *RMI – Remote Method Invocation*, <http://java.sun.com/products/jdk/1.1/docs/guide/rmi>, 1997.