Implementation Description

Laurence Cabenda
Bogdan Dumitriu
Richard Nieuwenhuis
Huanwen Qu

April 15, 2005

1 Changes to the design

In terms of design, we have not made significant changes, but we have slightly adjusted things here and there. The most important change (made at your suggestion) has been the separation of the GroupwareManagement interface in three distinct interfaces:

- **the UserManagement interface** - defines the methods to use for user creation and deletion, for logging in and logging out and for obtaining lists of the users of the system.
- **the CalendarManagement interface** - defines the methods to use for calendar management activities which cannot be part of the Calendar interface itself, since they don't pertain to a specific user's calendar. These methods are related to group appointments.
- **the MailManagement interface** - defines the methods to use for mail related activities. Again, it only contains those methods which do not pertain to a specific mail account or to a specific mail box.

We have also kept the GroupwareManagement interface itself, but only as a aggregation interface (i.e., it does not define any methods itself, but it aggregates the methods from the three mentioned interfaces by inheritance).

Most of the methods defined in these three interfaces (and, by aggregation, in the GroupwareManagement interface as well) should be seen as points of access to the system. In other words, most of these methods return some remote object which can then be used for particular functionalities. Because of that, their execution is only allowed if the user can provide a valid authentication token (see the Security section of our previous document for a discussion about this authentication token).

Apart from this structural change, the rest of the changes to the design were restricted to changing some of the method names, modifying their lists of parameters and adding a few setter and getter methods where we had omitted them. Another remark is that the implementation of the GroupwareManagement interface is indeed defined as a Singleton.

2 Implementation Details

In this section we give a quite detailed overview of our implementation, which should be helpful as a guide to browsing our code. Most of this project is written in Java, the only exception being the interoperability client, which was written in C. The organization of the project is as follows:

/src - contains the source code for the entire project
/config - contains a configuration file for the project
/lib - contains a JDBC driver for version 8 of the PostgreSQL database server
/sql - contains the script to be used for creating the needed database and tables
/classes - contains the compiled Java classes

As we mentioned in our previous document, our design is based on a 3-tier architecture, consisting of a database tier, a business logic tier and a client tier. The database tier consists of the PostgreSQL database server, which is a 3rd party product and shall not be described here. That leaves us with the business logic tier (which we, from now on, will call the server) and the client tier. We shall proceed to describing each of these two.

The server code

The server code can be found in the core Java package. It contains three main groups of classes: the so-called finder hierarchy, the remote interfaces and their implementations and the (data) holder objects, which are passed back and forth between client and server.

The first group of objects (the finder hierarchy) is used for mapping database rows to domain objects and making sure that there will never be more than one object in memory representing the same database row (in order to prevent inconsistencies in the data). The base class of this hierarchy is called *AbstractFinder*, and it provides some generic methods which leave only a few wholes to be filled in by the subclasses in order to specialize behaviour. It is also this class that ensures the uniqueness of mappings from database rows to domain objects. For this, the *Identity Map* pattern is used (see [Fowler2003], pp. 195-199). In a nutshell, this pattern implies the use of a Map (e.g., a HashMap) for storing all objects loaded in memory, indexed by some identifier (usually, the primary key of the database table). In this way, when an object is requested (based on its id) it can be checked if the object is already loaded and, if so, it will not be loaded again and instead the existing instance will be returned.

Subclasses of AbastracFinder are *UserFinder* and *AppointmentFinder*. These classes define the specific SQL code for retrieving objects from the Users and Appointments tables, respectively, and handle the loading of data into UserImpl and AppointmentImpl objects. In addition, they define a few other useful methods for finding and loading of certain database rows into objects. You will also notice a *CalendarFinder* class in this hierarchy. This is just pseudo-finder, since it doesn't get anything from the database, but it does ensure the Identity Map functionality for CalendarImpl objects (i.e., we don't want more than one CalendarImpl object per user to be active in memory at any time because this

can also lead to problems).

In order for this whole concept to work, we had to make sure that in all the server code objects were always created using methods of the Finder classes. Also, all of the Finder classes are defined as singletons, since otherwise their whole use is lost.

The second group of classes (and interfaces) you will find in the core package are those which define the remote objects that can are used in the system. We have the *UserManagement* and *CalendarManagement* interfaces aggregated by inheritance in the *GroupwareManagement* interface which define the methods for accessing the system and we have the *User*, *Calendar* and *Group* interfaces which define methods for finding and setting user details, adding/modifying/deleting single appointments and adding/modifying/deleting group appointments, respectively. The last four mentioned interfaces are implemented in classes with the same name followed by the *Impl* suffix.

The *GroupwareManagementImpl* class, apart from implementing the GroupwareManagement interface, also contains the main method which is used to start the server. Here, the singleton instance of GroupwareManagementImpl is retrieved and bound to an RMI registry which is started (or found) also in the main method. If so specified by a command line argument, the main method also loads the interoperability module, but this will be discussed separately later on. Relevant about this class is that:

- it keeps a authenticatedUsers hash table, indexed by the authentication token, where all the active users of the system are found. This table is also used for authentication when methods of this class are called.
- it unexports all exported remote objects associated with a user (namely, its User object and its Calendar object) when the user logs off.
- it uses functionality of the implementation (i.e., Impl-suffixed) classes which is generally not available in the interfaces for remote method invocation for proving implementations of the methods defined in the GroupwareManagement interface.

The *UserImpl* class is quite simple as it only offers a few setters and getters, most of which can also be called remotely. Apart from these, the class contains methods which handle the typical database operations (insertion, deletion and modification) for users as well as two methods for changing the password, which also work directly on the database.

The *CalendarImpl* class has methods for adding, changing and deleting appointments, all of which are implemented by writing directly to the database (i.e., there is no permanent in-memory list of appointments). The CalendarImpl class does the database work for administering user - appointment id mappings, while for the actual management of appointment data (such as location, duration and period), the class' methods delegate to their counterparts in the *Appointment* class. It should be noted that the Appointment class is actually part of the 3rd group of classes (i.e., the data holder classes), but it does have some extra code added to it for handling database operations.

Besides the handling of appointments, the CalendarImpl class also takes care of notifying the clients whenever appointments are changed. This is done in a

multithreaded manner for efficiency reasons. Last, but not least, the CalendarImpl class allows for retrieval of lists of appointments.

GroupImpl perhaps is the class that contains the most interesting business logic, namely the one for retrieving time slots which can be used for setting a group appointment. For this, it has some quite nice algorithms which are able to compute the periods of time which are free for all the users within a group of users, given some constraints. Besides that, it also contains methods for database operations which concern group appointments. In these methods, transactions are used in order to ensure correct registration of group appointments. This means that whenever the creation of a group appointment is requested, the selected time slot is checked again, to make sure that it is indeed available for all the users in the group, and then the appointment is actually registered for all the users. These operations are the ones done inside a transaction in order to ensure correctness.

The final group of classes is the one which contains data holder classes. These are the *Appointment* class, the *Period* class and the *UserData* class. They are usually just filled with data and send through the network.

For completeness, it should be mentioned that in the core package there are also the *ConnectionManager* class, which is a singleton class that provides connections to the database server to whoever needs them and the *ServerConfig* class, which is a wrapper over the configuration file.

The client code

We will naturally go into a lot less detail about the client code, since this mainly deals with displaying a GUI for the user to be able to use the facilities of the server. All the client-side classes are part of the client Java package.

The remote methods calls on the client side are done in the *GroupwareClient* class, which works as an interface between the GUI classes and the server. The GroupwareClient handles all the details of the remote calls, and only gives the GUI the raw data it needs (or some error messages if the case arises).

Another relevant class on the client side is the *ClientObserverImpl* class, an instance of which will be sent to the server so that the latter can notify the client by means of RMI whenever changes to the logged in user's calendar take place. Such an instance is registered by the GroupwareClient, right after the log in procedure succeeds and unregistered right before the log out procedure.

Of course, most of the remaining client code consists of the GUI code, which is organized in a classical Model-View-Controller style, with the view and the controller in the (quite large) *ClientController* class (and a few other helper classes) and the model distributed between *ClientModel* and *ClientAppointmentModel*.

The interoperability code

The code specific to the interoperability module is separated into the core.interop (the server side) and client.interop (the client side) packages. The module provides just some limited functionality, namely the retrieval of the appointments of a user. The whole interoperability module is based on CORBA, and the IDL definition for it can be found in the Calendar.idl file, which you can find in the src/ directory.

The class created by us (in contrast to the automatically generated classes) on the server side is *core.interop.CalendarImpl* which gets the list of requested appointments from the main server module and transforms them in such a way that they fit into the automatically generated framework.

The counterpart on the client side is the *client.interop.CalendarClient* class which provides a text based interface for the user to specify his login name and password, then gets the appointments from the server by using CORBA for the remote call and then displays the retrieved appointments.

A client with a similar functionality has been written in C as well, using the ORBit2 2.12.1 CORBA ORB implementation. This client can be found in the src/c-client directory. It simply illustrates the interoperability concept even more by connecting to the Java server from a C client by using CORBA, getting and displaying the list of appointments.

3 Security

In what security of our system is concerned, we have ensured a certain degree thereof by the authentication mechanism together with the fact that clients cannot get access to objects they are not allowed to, both of which are described in our previous document. Naturally, without any kind of encryption employed by our system, the authentication step can quite easily be breached by simply monitoring the network traffic and thus retrieving the (clear text) passwords. Furthermore, the privacy of the users is also in jeopardy without encryption, since their entire calendar can be viewed by the use of network sniffers. Last, but not least, an attacker could spy the network and get references to the objects to which normally access is not allowed and then use these references to actually change a user's details or appointments. The natural solution to all these problems is, of course, encryption of the communication between the client and the server, either at the network level (e.g. by using IPSec) or at the application level (e.g. by using RMI over SSL, if such an possibility exists).

Since this is a groupware system, we would normally expect it to run within an organization, which means that additional security could be provided by disallowing access to the machine running the server from any outside hosts. This would still leave us with the problem of internal security, but it would nevertheless be a welcome addition.

Of course, a virtual attacker could also connect to the RMI server directly and, with sufficient knowledge about the protocol, he might be able to get access to the objects running in the server even without previously being in the possession of a reference to them. Encryption wouldn't necessarily be able so solve this problem (unless we decide to give each user a key and only allow access to the keys we have distributed to the users), This leaves us with the solution of protecting access to *all* the methods which can be called remotely, either with a password or with the authentication token which is already in use.

4 Instructions for running the software

Please see the README file that comes with our source code.

References

[Fowler2003] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, 2003.