

A pattern system for the development of collaborative applications

Luis A. Guerrero^a, David A. Fuller^{b,*}

^a*Depto. de Ciencias de la Computación, Universidad de Chile, Chile*

^b*Computer Science Department., Pontificia Universidad Católica de Chile, Computer Science, Santiago, Chile*

Received 14 March 2000; revised 25 July 2000; accepted 2 February 2001

Abstract

Collaborative applications provide a group of users with the facility to communicate and share data in a coordinated way. Building collaborative applications is still a complex task. In this paper we propose a pattern system to design basic aspects of data sharing, communication, and coordination for collaborative applications. These patterns are useful for the design and development of collaborative applications as well as for the development of platforms for the construction of collaborative applications. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: CSCW; Groupware; Design patterns; Collaborative applications construction

1. Introduction

Collaborative systems (also named *groupware*) are software that support people working in groups. As Ellis, Gibbs, and Rein stated, the goal of collaborative systems is to assist groups in communicating, collaborating, and coordinating their activities [8].

Over the past two decades, researchers have been working to develop collaborative systems to increase the productivity of work groups [18]. However, building groupware is a complex task because it involves issues that do not appear in single-user systems, such as human-to-human communication, group dynamics, users' social roles, group memory, and other organizational and social factors. Most collaborative applications are composed of a set of inter-related modules, including access control, notifications, user management, group interface, packet distribution, data storage, data views, work sessions, awareness information, and user communication [7,8].

At present, there is some research and several toolkits available for the development of collaborative applications [3,5–7,12,14,16,19,21,25–29], however there is little investigation on design techniques (see Ref. [17] for workflow design techniques). As we show in the next sections, there are common features in most collaborative applications that must be supported by toolkits. In fact, some of the currently

available toolkits support some of these features, but no single toolkit supports all needed features, forcing developers to implement these themselves.

We are focused in the design and implementation of collaborative applications. Particularly, we are focused in mechanisms to manage work sessions, access control, and communication among users, information objects, views and floor control policies.

This paper describes a pattern system for the development of collaborative applications. Section 2 describes some aspects related to the construction of collaborative applications. Section 3 describes currently available collaborative tools and applications. Section 4 elaborates on the four patterns of the developed pattern system. Section 5 explains the relationships among the patterns when collaborative applications are designed. Finally Section 6 presents our conclusions.

2. Construction of collaborative applications

From the perspective of the construction of collaborative applications, it is possible to define four different generations showing the development in time of this process.

2.1. First generation: monolithic systems

During the development of the first collaborative systems, the construction and integration of the application modules had to be done by the programmers. Due to this, this process required a large effort. We define this scenario as the first

* Corresponding author.

E-mail addresses: luguerre@dcc.uchile.cl (L.A. Guerrero), dfuller@ing.puc.cl (D.A. Fuller).

generation of collaborative systems (from their construction point of view). Construction and testing of this kind of systems took a long time, since there were no development tools to facilitate the process. One had to start from scratch, and only the C programming language was used. Later on, some tools appeared.

2.2. Second generation: tools to extend languages

Perhaps the most known tool to support the development of collaborative applications is GroupKit [21]. GroupKit is an extension to the Tcl/Tk language, adding various commands to allow synchronous collaborative-shared sessions. Another well-known tool to build collaborative applications is the NCSA's Habanero system [5]. Habanero is a Java language *extension* to 'share' monouser applications. Other tools extending the Java language are '*Java Collaborative Environment*' (JCE) [JCE], '*Java-Enabled Telecollaboration System*' (JETS) [26] and '*Java Shared Data Toolkit*' (JSDT) [JSDT].

We say that the systems built using the type of tool that extends one language to provide facilities to implement collaboration mechanisms, correspond to the second generation of collaborative systems. With this approach, the programmer is provided with facilities to improve efficiency on his work. However, he is restricted to a particular language. This approach of extending the functionality of language is commonly used in Web applications. Due to HTML restrictions, it has been extended to accept CGIs, JavaScript code, applets, *servlets*, *oblets* [30], *displets* [31], '*Java Applets Made Multiuser*' (JAMMs) [JAMM], etc. When using these HTML language 'extensions', it is possible to build collaborative applications on top of the Web. Some of these tools allow code reuse, which could also be considered as members of a *third generation*.

2.3. Third generation: object oriented platform

In the last three years, the literature has reported some frameworks composed by classes of objects encapsulating specific functionality of collaborative applications. For example, the '*Notification Service Transfer Protocol*' system (NSTP) [6] to build synchronous collaborative applications on the Web, based on the events notification service. '*Group Communications*' (GroCo) [29] and Meta Web [28], also support the development of synchronous collaborative applications on the Web.

We say that applications based on these kind of object oriented frameworks correspond to a *third generation* of collaborative applications. The main advantage with respect to the previous generation is the code reuse, since functionality is encapsulated in classes of objects. The development of these platforms has been an important contribution in the last few years. For example, the 1997 ECSCW Conference ('*European — CSCW*'), organized a workshop around this subject ('*Object-Oriented Groupware Platforms Workshop*', OOGP'97). Until now, and according to the way

that collaborative applications are built, they can all be considered in one of these three generations.

2.4. Fourth generation: distributed objects and components

Recent advances in the distributed objects' area (CORBA, DCOM and RMI) and components based programming (Visual C++, Visual Basic, Delphi, JavaBeans, etc.), define a *fourth generation* of collaborative applications, built using these elements. The main advantage of this new generation is that components, by definition, are language independent. Also, components provide a visual programming environment. Microsoft's Collaboration Data Objects (CDO) is an example of this new generation of tools to build collaborative applications.

With the patterns proposed in this paper, it is possible to build a set of components to provide basic functionality to build collaborative applications in languages that allow the use of components.

From a commercial point of view, the four most used tools to build collaborative applications, mainly workflow applications, are provided by Lotus (Domino/Notes), Microsoft (Exchange/Outlook), Netscape (SuiteSpot) and Novell (GroupWise). The two main Microsoft tools to develop collaborative applications are Microsoft Outlook (at the client side) and Microsoft Exchange Server (at the server side) [32]. Outlook supports the ability to manage information (e-mail messages, appointments, contacts, and tasks) and share it throughout an organization. Outlook also includes a development environment that allows to write collaborative applications. Exchange Server supports communication, information sharing, and workflow services using Internet standards and protocols. The use of these two systems allows the construction of applications based on messages sharing information stored in databases. Other applications, such as NetMeeting, allow audio and video transmission, as well as sharing information with other users by using whiteboards, chats and group calendars. The facilities provided by Microsoft tools are mainly oriented towards workflow applications. However, it is possible to develop other type of applications using programming environments such as Visual C++, Visual Basic and ASP. Starting from the Exchange version 5.5, Microsoft included a number of objects called CDO. CDO is a scripting-object library that developers can use to design collaborative applications on both the client and server side. CDO supports the creation of workflow and scheduling applications.

The tools provided by Lotus to create collaborative applications are Domino (the server) and Notes (the client). The Domino server provides support to build distributed applications using CORBA and IIOP. Domino and Notes support the construction of workflow and collaborative applications based on messages.

Netscape's SuiteSpot provides functionality to support the development of collaborative applications by providing

Table 1
Common features in some CSCW toolkits

| | GroupKit | Habanero | JSDT | NSTP | GroCo | MetaWeb | COAST | CBE | Artefact | CDO | Mushroom |
|--------------|----------|----------|------|------|-------|---------|-------|-----|----------|-----|----------|
| Sessions | X | | X | | X | X | X | X | X | X | |
| Users | X | | X | X | X | X | X | X | X | | X |
| Roles | X | | | X | X | | | X | X | | X |
| Messages | X | X | X | X | X | X | X | X | | X | |
| Objects | | | X | X | | X | X | X | X | | X |
| Repositories | | | | X | | X | X | | X | X | X |
| Views | | | | | | | X | | | X | X |

a set of servers, mainly workflow and message based applications. Novell's GroupWise is a messaging system that offers some communication and collaboration capabilities, such as document management capabilities, calendar options, group scheduling, automated workflow, task and document management, rules-based message management, and electronic discussions.

These four commercial alternatives are based on messages, being very useful to develop workflow applications. However, they do not offer major facilities to build other types of collaborative applications nor provide support to design them.

3. Design patterns in collaborative applications

A *design pattern* [9] is a recurring solution to a problem in software design. Design patterns encapsulate experience, provide a common vocabulary for computer scientists across domain barriers, and enhance the documentation of software designs [1]. In general, a pattern documentation describes a *context* within which we can use the pattern, the *problem* the pattern solves, and the *solution*. Moreover, we can describe the *forces*, *implementation*, *examples*, and *related patterns* as well. Thus, according to Sane [22] a pattern paper tells the story of design decision: in some *context*, if one faces a *problem* whose *solution* must satisfy certain requirements or *forces*, then one must use the *solution*; but keep in mind that it has *consequences*, and certain *implementation* issues will be faced.

Individual patterns can be interwoven within *pattern systems* [4] that describe how they are connected, how they complement one another, and how software development with patterns is supported. Pattern systems aid developers in communicating architectural knowledge, teach new design paradigms or architectural styles, and train new developers in avoiding costly traps and pitfalls which have been traditionally learned through experience [23].

Focusing on the identification of design patterns for collaborative applications, we studied some tools for the construction of these applications. These tools were: GroupKit [21], NSCA Habanero [5], JSDT [JSDT], Lotus NSTP [6], GroCo [29], MetaWeb [28], 'COoperative Application System Toolkit' (COAST) [25], 'Collaboratory Builder's Environment' (CBE) [14], Artefact [3], Mushroom [12],

DistView [19], Oval [16], 'Java-Enabled Telecollaboration System' (JETS) [26] and Dagora [27].

Moreover, we studied a number of collaborative applications, which have demonstrated a certain degree of success. These included Microsoft NetMeeting [NetMeeting], GMD 'Basic Support for Cooperative Work' (BSCW) [BSCW], Instinctive Technology Inc. eRooms [eRooms], Netopia Inc. Netopia Virtual Office [Netopia], TeamWave Software TeamWave Workplace [TeamWave] and Lotus Notes [Notes].

After examining previous development tools and applications, various common concepts were identified that can be applied to the patterns definition. As shown in Table 1, these concepts are: sessions, users, roles, messages, objects, repositories, and views. The *sessions* (also called *work sessions*, *groups*, or *conferences*) maintain information about the users that interact in an asynchronous or synchronous way through a collaborative application. The *users* (or *collaborators*) are the members of the work group. Every user can have different access rights according to their *roles*. *Messages* (*notifications* or *events*) are used to communicate among users and also among application modules. *Objects* (or *information objects*) are data produced by the users. Most of the time, we need to store objects' attributes, which are information about the information objects (or meta-information) such as the owner of the object, the creation date and time, and previous versions. The objects are stored in *repositories*. Finally, a *view* shows part of the objects in a repository.

In addition to these seven common features of the collaborative applications toolkits, two more can be included: *environments* and *floor control policies*. The *environments* organize and coordinate multiple working sessions, or multiple user groups working on the same collaborative application. The environments allow sharing collaborative applications among many user groups. The *floor control policies* define the strategies used to manage the shared resources. The above-mentioned components are explained in the following sections and are presented as patterns for the design and development of collaborative applications.

4. The top patterns

Using the above nine components, plus a *broker* pattern

[4], we designed and built a prototype pattern-based framework for the development of collaborative applications named ‘Ten Objects Platform’ (TOP) [10]. The TOP platform provides support to the construction of collaborative applications, adding all the components mentioned in Table 1. TOP also includes the components environment, floor control and broker.

All of these components have been organized in a pattern system composed of four main patterns for the design of collaborative applications. The next sections describe these four patterns.

4.1. Pattern 1: group memory

In general, all collaborative applications must store and share data created by the users. In addition, collaborative applications need to provide views of the information, because various kinds of users can have different perspectives on the same data. All data users create form the *group memory* of the application.

4.1.1. Problem

How to provide users with the necessary tools that will allow them to consistently retrieve objects created and saved by other collaborative application users? How can users share data? How can views of the group memory be provided?

Users produce data as result of collaborative work; thus, these data must be stored. Users within the group must share the data produced. A user can recover, remove or modify a data object belonging to the group. Sometimes it is necessary to organize information in several data repositories. In order to provide data awareness it is necessary, in most applications, to maintain information on each data object such as the user who creates it, the creation date and time, the object type, some keywords to aid in searches, and previous versions of the objects. Certain applications need

to deny access to data among specific users or during specified time periods. Some data need to be hidden from certain users. Some users require seeing data in a different way (larger, different order, etc.).

4.1.2. Solution

Provide information repositories that can be managed from the applications. Furnish the necessary methods to manage the data objects contained within these repositories so that they can be shared among the users. Provide an *information object* for each data object. This object or meta-object should contain attributes or information such as the object’s creation date and time, its owner or creator, the object type, previous modifications, keywords, and previous versions. Several data views can be generated for each repository. Users (or the application) select the view they wish to use in order to show the data graphically.

The use of this pattern is convenient when collaborative application need to manage group memory information. In simple applications a single repository and view is sufficient.

4.1.3. Static structure

Four collaborator classes form the *group memory* pattern: repository, view, meta-object (or attributes) and object. The *repository* stores the objects produced by the users. A *view* shows part of the objects stored in a repository. There are two kinds of views: ordered list and query. The ordered list has a pre-defined ordered set of objects. The query view is a SQL-based template, with a *refresh* method that shows the objects matching the query. The *attributes* provide information about each object. The *objects* are the data produced by the users. Fig. 1 shows the UML diagram with relationships between the components of the pattern.

4.1.4. Dynamic structure

The main scenario is how to create new objects and how

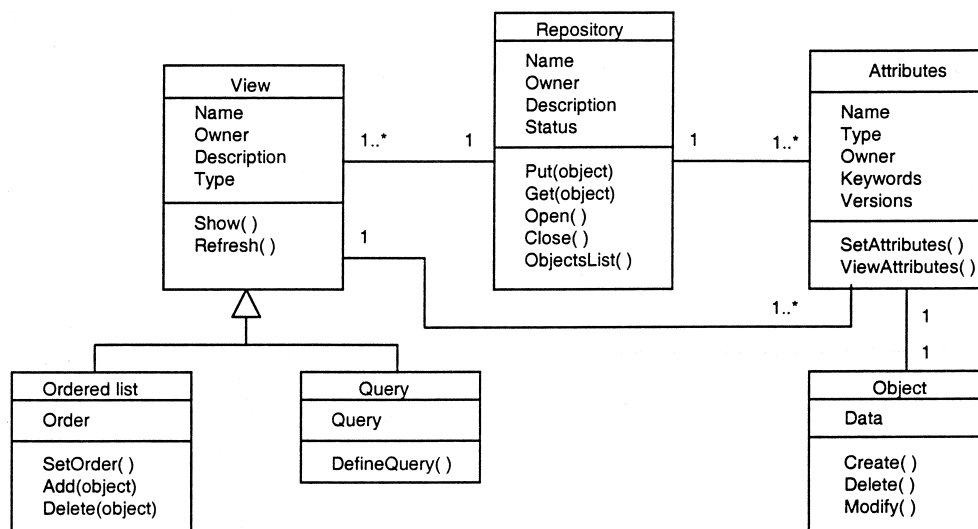


Fig. 1. Classes diagram of the *group memory* pattern.

to link these objects with the respective repositories. In this action the repository must refresh the necessary views. This scenario is shown in Fig. 2.

In Fig. 2 diagram, once the object is created, the attributes of the object are defined. After that, the real object is associated with the repository. Finally, all the views add the object.

4.1.5. Implementation

When the developer use this pattern, he or she can organize the information in several repositories. The developer can define various views for each repository according to the application.

Each repository should have information about its name, owner or creator, description, and state ('open' whether the information is available or 'closed' if it is restricted). A protocol should exist to check if a repository exists, to offer a list of application repositories, to check if an object is available within the repository, to close the repository, to open it, and to erase it. Also required is the ability to delete an object that is inside the repository, to place an object inside it, to recover an object, and to provide a listing of the objects it contains.

The *meta-object* or *attributes* should contain information with its logical name (of the meta-object) and the related physical object (the real data object), the object type, the owner or creator, the last user who modified it, the creation and/or last modification date and time, keywords, and previous versions. It should also have methods to check if an object exists, to recover the real name of an object, and to know an object type.

The *view* object type should have information with its name, its owner or creator, the repository (or repositories) it belongs to, the view type (an ordered list, or a SQL-like query), its description, the date and time of its last modification, the ordered objects list (in case of an ordered list view), and the query (in case of a SQL query view). It should have methods to store and to delete an instance, to check if a view

exists, to refresh an order (in the case of an ordered list view), and to refresh the view (in case the objects of the repository change).

4.1.6. Known uses

In some *brainstorming* systems a workgroup presents a grouping of ideas or proposals. Once the presentation phase of ideas ends, a voting may be taken to choose the best proposals. In this case the generated ideas can be contained within a repository while the votes are maintained separately in another. The 'object's repository' idea exists in systems like NSTP, MetaWeb, COAST, CBE, Artefact and Mushroom, and in some collaborative applications such as BSCW and eRooms.

If queries on objects such as video, images, or sounds are desired it is convenient to have additional information in a *meta-object* with keywords that describe the object. Furthermore, we may want to identify which user created it, when it was last modified, or when it was completed (data awareness). When sending 'message' type objects to other users it is necessary that the receiving user knows when it was sent and which user sent it. If a user modifies or erases an object created by another user, it is necessary to register this action. Such objects are present in systems like JSDT, NSTP, COAST, CBE, Artefact, and Mushrooms.

In a chat application users receive messages in the same order they were sent (ordered view). Systems such as COAST and Mushroom utilize the idea of *views*.

In a collaborative text editor, each document could be stored in a *repository*. Each element of the document, such as paragraphs, titles, figures, tables, etc. could be modeled as a *data object*. In order to provide data awareness, each data object could be associated an *attributes* object. This object could contain information about the user who created it, date and time of creation, the last modification, as well as previous versions. Each document of the editor could be showed by the application as a *view* of type *ordered list*.

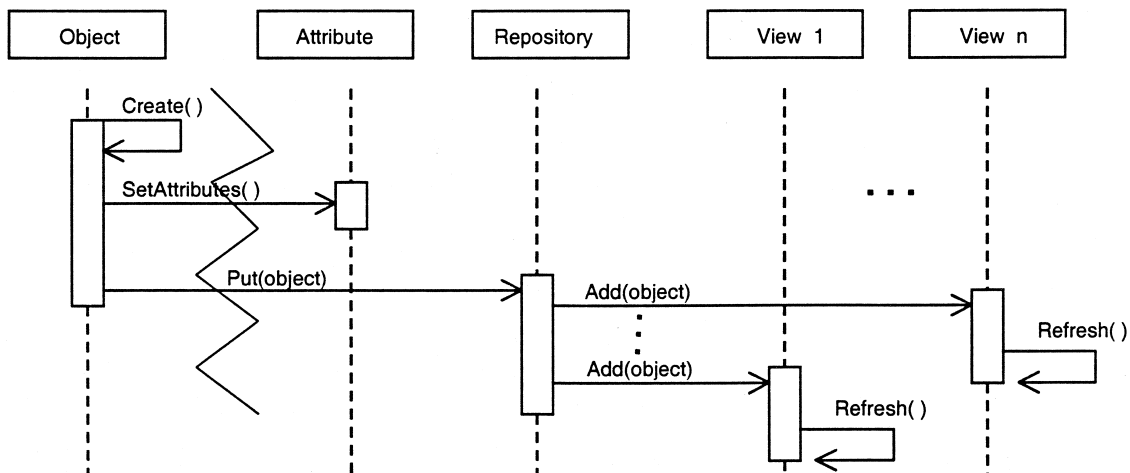


Fig. 2. Create an object in a repository.

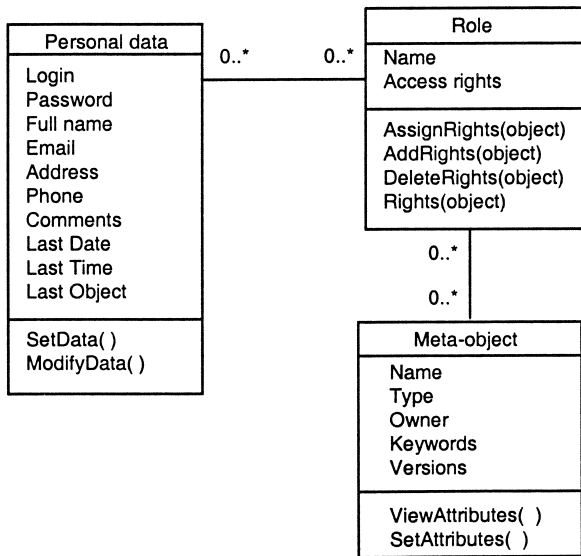


Fig. 3. Collaborator pattern classes diagram.

4.1.7. Related patterns

The *shared repository* pattern [13] is an architectural pattern that defines a communication model for software components based on shared memory. The *observer* pattern [9], also known as *publisher-subscriber* [4], defines a one-to-many dependence among objects such that when an object changes, all their dependencies are notified and updated automatically.

4.2. Pattern 2: collaborators (users or group members)

In a collaborative application several users make up a workgroup. It is necessary to maintain certain personal information for each user. In many collaborative applications users have different access rights to the information.

4.2.1. Problem

How to manage the information of the workgroup's users? Not all users have similar access rights to information. There are restricted data and operations. How do we manage users' access rights?

When a user enters an application it is sometimes necessary to check their name and password. To provide user awareness, it is necessary to maintain certain information about each user so other group members may identify them. Most collaborative applications have different types of users (roles) with different access rights over data. Certain operations (erase, view, modify, recover) or processes (backup, print, etc.) are restricted from certain users.

4.2.2. Solution

A *user* object can have all the necessary personal information for each of the users participating within the

application. Assign *roles* to the users, which define each user's role: their access rights over the objects of a repository, and views. This pattern is useful when collaborative applications have several users with different access rights. For applications with a single kind of user, this is not necessary.

4.2.3. Static structure

Three collaborator classes compound this pattern: *personal data*, *role* and *meta-object*. *Personal data* have personal information for each group member, for example, login, password, full name, postal address, e-mail, phone, and any other information necessary to provide adequate awareness information. The *role* component has information about the right access and the respective list of objects. The *meta-object* component has additional information to provide data awareness (this component was defined in the above pattern). Fig. 3 shows a UML diagram with the relationships between these components.

4.2.4. Dynamic structure

The main scenario (see Fig. 4) occurs when a user wants to modify an object. The scenarios to create, delete and view an object are analogous.

In Fig. 4, before making the modification, the access rights are checked. After that, the access rights list is returned.

4.2.5. Implementation

The *user* object can contain the user's name, password, full name, email, postal address, telephone number, photo, comments, last connection date and time, last object modified, and the role. It also can have methods to store an instance of the object, to check if a user exists, to request a list of users and to erase users.

The *role* object must maintain information with its name, list of views it has access to and repositories list describing the types of access rights for each one. Access rights are applied to repositories, although it is possible to refine the

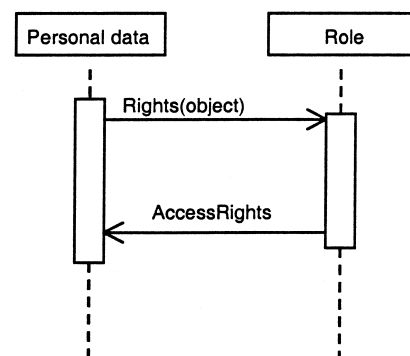


Fig. 4. Modifying an object.

kind of rights, which apply them to specified repository objects in a finer granularity outline. The *role* should implement methods to check if a role exists, to check if a user has enough rights on a repository or view, to add views and repositories with appropriate access rights, and to request a list of roles in certain sessions.

4.2.6. Known uses

Many restricted applications request a user's name and password. To provide user awareness some applications display the picture and the user's full name. Most of the systems mentioned at the beginning of this section use the idea of 'users'.

Within most collaborative text editors two types of roles can be found, defined as *readers* and *writers*. Readers cannot add text; they can only make comments on the text written by writers. Another widely used role in collaborative applications is the work sessions *coordinator*. Different user's roles can be defined in GroupKit and CBE.

4.2.7. Related patterns

The *role object* pattern [2] defines an object which can change its role dynamically. The *persons* and *roles* pattern [24] allows users to participate inside a system with different roles which they can change dynamically. The *documents and roles* pattern [24] shows the relationships between documents and people's roles within a system.

4.3. Pattern 3: floor control policy (or floor control)

Some objects require mutual exclusion, meaning they can be used by only a single user at the same time. For these objects, a floor control policy should be implemented defining the way in which the object is assigned to the user who requests it.

4.3.1. Problem

How to provide floor control on objects? What assign-

ment policy can be applied that decides which control is granted to which user?

Some objects restrict their use to a single user. If a user requests a floor control object, and the object is free, the object is assigned and is marked as 'in use'. If a user requests a floor control object and another user is using this object, the new user is added to a waiting list. When a user releases a floor control object and if that object has a waiting list, the object is assigned to the following user in the list, according to the current assignment policy.

4.3.2. Solution

A *floor control policy* object type is associated with objects with restricted access. This new object manages information and coordinates the use of the restricted object. When a user requests this object, the floor control is checked. Each floor control object has an assignment policy.

This pattern is useful in applications that require mutual exclusion for some objects, and need to define an assignment policy.

4.3.3. Static structure

Two object classes compose this pattern: *floor control* and *meta-object*. Fig. 5 shows the UML classes diagram of this pattern.

4.3.4. Implementation

Using this pattern, the developer of collaborative applications need not worry about the implementation of mutual exclusion over shared objects. He or she also need not worry about the assignment policy among the users that require the objects.

The floor control object should have information with its name, the repository within which it is stored, the user who has the control, and the users on the waiting list. It should also have methods in place to create and to erase a floor control object, to check if a floor control object exists, to

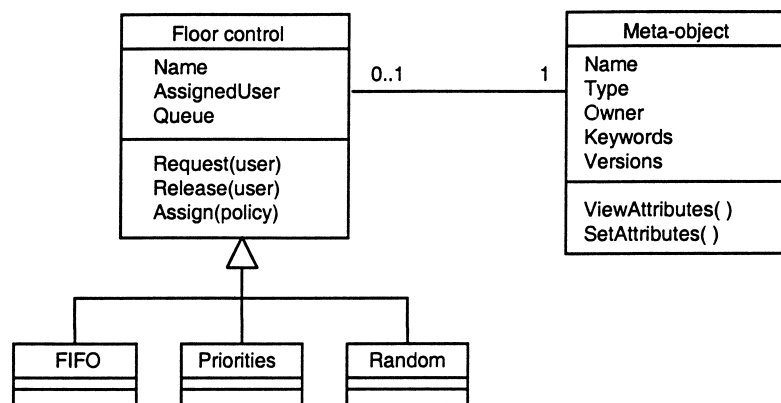


Fig. 5. Floor control policy pattern classes diagram.

return a list of floor control objects in a repository, and to request and release the control.

4.3.5. Known uses

Videoconferencing is possible when all the users see the image of all the other users at the same time, but with users speaking one at the time. It should have a specific kind of ‘microphone’ object that can be passed by means of an assignment policy (as FIFO) among the users, which affords a user with the ability so speak.

4.3.6. Related patterns

Guillermo Licea and Jesús Favela describe in [15] a similar floor control object detailing the different assignment policies in passing such a control.

4.4. Pattern 4: collaborative session (workgroup, work session or conference)

By definition, in all collaborative applications work is performed by user groups in work sessions. These sessions can be both synchronous and asynchronous. It is necessary, therefore, to manage these work groups, checking users who enter and leave them. Users working in a specific session do not view the work of other users working in a different session of the same collaborative application. One collaborative application can have many work sessions. Each session may have different users working on different repositories, which means that some groups do not interfere with others. This allows the re-use of an application so that several groups may use it simultaneously.

It is common that during a work session users need to send objects or messages to one another. A user can send an object to all users, to connected users, to a specified group of users, or to a single user. An application can send events, commands, or notifications to other users or even to other applications.

4.4.1. Problem

Several workgroups may work within the same application without interfering with one another. How do we provide uniform access to collaborative applications, identifying users among corresponding sessions? How do we manage and coordinate workgroups in collaborative applications? How can we make other work sessions transparent? How do we send objects to other users within the workgroup?

Collaborative applications users work in groups. One application may support several work sessions composed of different users working in multiple data repositories. The work of a session member should not interfere with the work of another session’s members. In many situations it is necessary to know which users belong to which workgroups. It is sometimes necessary to know which workgroup users are working at that specific moment, in order to interact synchronously. Users that do not belong to the workgroup cannot see or modify the data. New users can

be defined as members of the group. Users that belong to the group can leave. Sometimes users must send objects or messages to other users within the workgroup. Sometimes users must send objects or messages to users connected only at that moment, or send objects to all the users within the group (connected and not connected). Sometimes the system must send events, commands, messages or notifications to connected (or disconnected or both) users. All users among different sessions should have a uniform way to enter a collaborative application. The system should be able to identify in which session individual users may work.

4.4.2. Solution

Maintain a list of all users belonging to the workgroup, as well as a list of users currently working (connected). Request the name and password of each user requesting to work with the shared data and validate these. Define communication channels in order to send objects or messages to other users.

An *environment* object can store information on all work sessions or different users’ groups which are using the collaborative application but working on different areas.

This pattern can be useful for applications that need to maintain a list of users with specific properties (personal data, passwords, etc.) and send objects or messages among them. In addition, this pattern allows sharing an application among several groups of users.

4.4.3. Static structure

Four components compose the *collaborative session* pattern: *environment*, *session*, *channel*(or *communication channel*) and *user*. The *environment* component allows a collaborative application to have several work groups. The *session* component manages the users’ information, including the connected and disconnected list of users. The *channel* component has a list of users that belong to the work session. It also has technical information (such as the host name and port) in order to send and receive messages. Using this component, it is possible to send objects (messages, events, notifications and information objects) to the session users. Finally, the *user* component has personal information about the users. Fig. 6 shows the relationships among the components of this pattern.

4.4.4. Dynamic structure

The main scenario is when a user sends an object to other users. Fig. 7 shows this situation.

In Fig. 7, a user sends a message to the work session. After that, the session sends the message to the respective channel. Finally, the channel sends the message to all the users in the list.

4.4.5. Implementation

This pattern provides communication among the members of the work group. The pattern allows objects to be sent inside a collaborative work session.

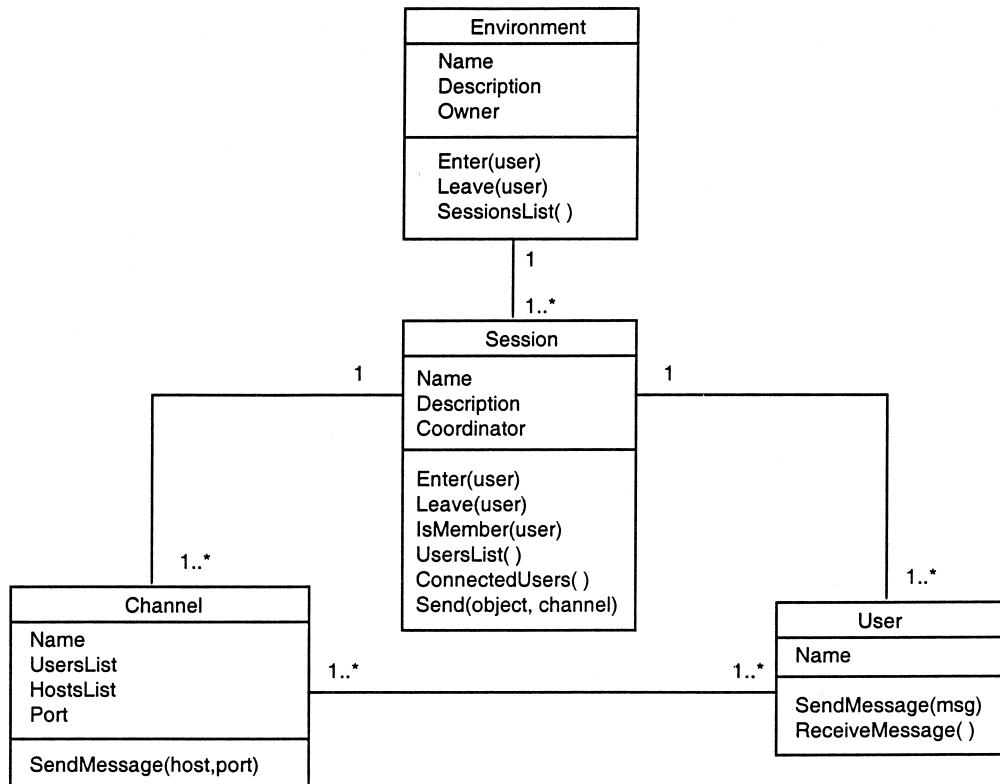


Fig. 6. Collaborative sessions pattern class diagram.

Each session should be furnished with a name, description, a coordinator, and a user’s list. A continuously updated user’s list should be maintained. There should be a method to list all the sessions. Protocol should exist requesting users to identify themselves as either a member or coordinator of a session, and to know if a user is currently connected (working) in the session. A protocol should exist for a user to enter a session, and to leave it.

An *environment* object should maintain information with

its name, description, owner or creator, and work sessions list (each session maintains its respective repositories list). When a user enters an environment the environment checks and enters the user into the corresponding work session.

The *channel* object has two basic methods: to send an object to all users within the session or to send an object to a user list. The same *channel* object can be used by the system to send events to other users’ applications.

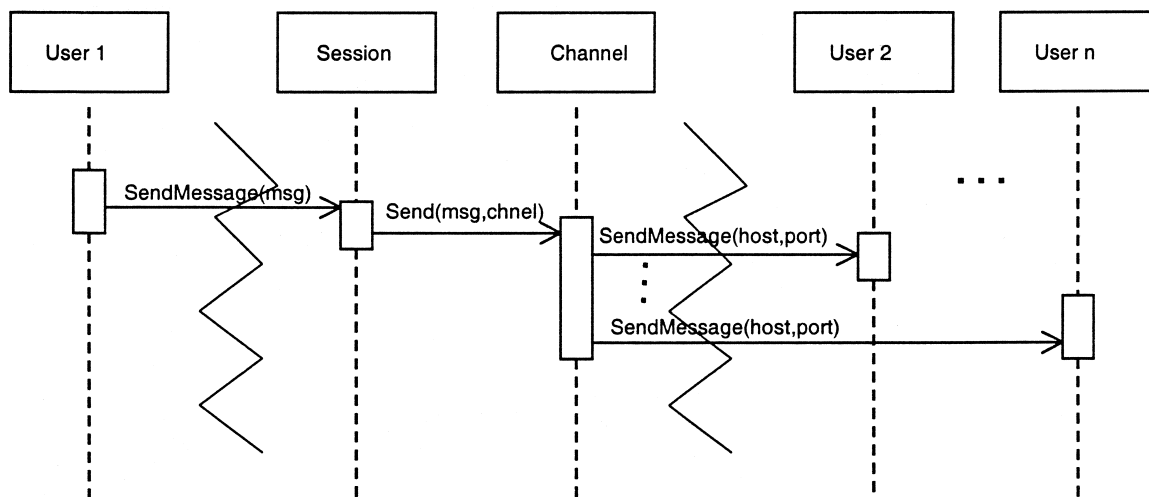


Fig. 7. Sending messages through the channel.

4.4.6. Known uses

Conference systems such as Microsoft NetMeeting allow several users to simultaneously share a group of applications. Users can enter work sessions or leave them. Other users can be invited to participate in the work session. Similar situations occur with the applications we constructed using GroupKit, JSDT, GroCo, MetaWeb, COAST, CBE and Artefact.

When several people participate in a chat application, and if a user sends a message, all users currently connected in the session receive it. The ‘channel’ idea is present in systems such as GroupKit, Habanero, MetaWeb and COAST.

4.4.7. Related patterns

The *abstract session* pattern [20] allows a server object to maintain the status of each client it serves. The *mediator* pattern [9] defines an object which encapsulates the way in which a group of objects interact.

The *publisher-subscriber* pattern [4], also called *observer* [9], maintains a synchronized state of cooperating components. A publisher object notifies subscribers about any change in the state.

5. A pattern system for collaborative applications design

The previous nine patterns define a *pattern system* for the design of collaborative applications. Fig. 8 describes the way in which these patterns were related in the design of the TOP platform.

In the TOP platform, the *environments* define the general context of the collaborative applications. They allow several workgroups to share an application. Through the *sessions* the environments manage multiple user groups that are working with different data. The sessions can send messages

or events to the users through the *channel*. The sessions define mutual exclusion on certain objects throughout the *floor control*. The information about each user in a session is defined in *user*. Each user has certain rights, or data access rights, which are defined through *roles*. The shared information is composed of *objects*. These objects are stored in data *repositories* and they can be seen through *views*.

6. Discussion

This paper presents a pattern system for the design and development of collaborative applications. The nine patterns which compose the system are: *environments*, *sessions*, *roles*, *channel*, *floor control*, *repositories*, *view* and *objects*. Many of these patterns can be clearly identified among many of the tools for the development of collaborative applications, as well as within most of the collaborative applications themselves.

The *environment* pattern allows several user groups to simultaneously share an application without interfering with others groups. This allows the transformation of *mono-group* applications into *multi-group* applications. The *environment* and *session* patterns allow re-use of the applications, as it is sufficient enough to create sessions for other groups to use the applications.

Some of the main features of collaborative applications can be designed from these patterns, including, for example, shared data (*repository*, *objects*, *session*), communication among users (*channel*), coordination (*floor control*, *users*, *roles*), group memory (*repository*, *objects*), users (*users*), roles (*roles*), data awareness (*objects*, *repositories* with versions), user awareness (*users*, *sessions*), data views (*views*), work sessions (*session*), access control (*roles*, *sessions*), floor control mechanisms (*floor control*), notifications (*channel*), events, and messages (*channel*).

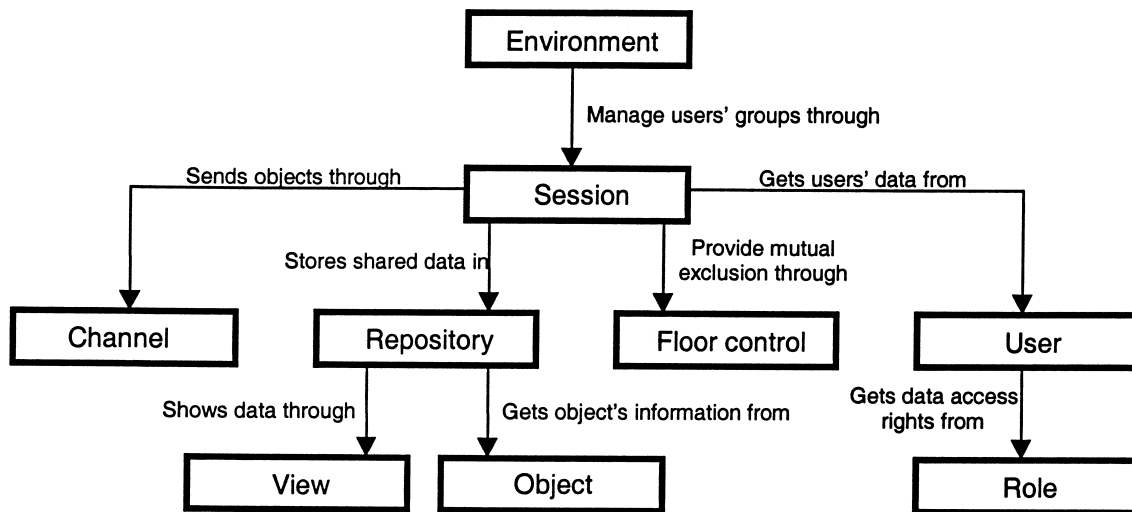


Fig. 8. The pattern system of TOP.

These patterns were used in the TOP platform design. TOP's pattern system illustrates how to solve most of the problems posed when designing and building collaborative applications. TOP's pattern system many also be used for the design and development of distributed systems.

The current TOP platform version is the result of several iterations, which is typical for pattern-based framework development. Our experience suggests that the pattern-based architecture of TOP makes collaborative systems development easier.

Acknowledgements

The work has been partially supported by the Chilean Science and Technology Fund (FONDECYT) under grant No. 1980960.

References

- [1] E. Agerbo, A. Cornils, How to preserve the benefits of design patterns, Proceedings of OPSLA'98, Vancouver, Canada, 1998.
- [2] D. Bäumer, D. Riehle, W. Siberski, M. Wulf. The role object pattern, Proceedings of the PLOP'97, Monticello, Illinois, USA, 1997.
- [3] J. Brandenburg et al., Artifact: a framework for low-overhead web-based collaborative systems, Proceedings of CSCW'98, Seattle, Washington, 1998.
- [4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, Wiley, New York, 1996.
- [5] A. Chavert, E. Grossman, L. Jackson, S. Pietrowics, C. Seguin, Java object-sharing in Habanero, Communications of the ACM 41 (6) (1998) 69–76.
- [6] M. Day, J. Patterson, D. Mitchell, The notification service transfer protocol (NSTP): infrastructure for synchronous groupware, Sixth International World Wide Web Conference, Stanford, California, USA, 1997.
- [7] B. Eiderbäck, L.A. Jiarong, Common notification service. Proceedings of the OOGP'97, The ECSCW'97 Workshop on Object Oriented Groupware Platforms, Lancaster, UK, September, 1997.
- [8] C.A. Ellis, S.J. Gibbs, G.L. Rein, Groupware: some issues and experiences, Communications of the ACM 34 (1) (1991) 38–58.
- [9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994.
- [10] L. Guerrero, D.A. Fuller, A web-based OO Platform for the development of multimedia collaborative applications, Decision Support Systems 27 (3) (1999) 241–254.
- [11] R. Johnson, Documenting frameworks using patterns, Proceedings of OOPLA'92, Vancouver, Canada, 1992, pp. 63–76.
- [12] T. Kindberg, A. Mushroom, Framework for collaboration and interaction across the internet, Proceedings of the ERCIM Workshop on CSCW and the Web, Sankt Augustin, Germany, February 1996.
- [13] P. Lalanda, Shared repository pattern, Proceedings of PLOP'98, Monticello, Illinois, USA, 1998.
- [14] J. Lee, A. Prakash, T. Jaeger, G. Wu, Supporting multi-user, multi-applet workspaces in CBE, Proceedings of CSCW'96, Boston, Massachusetts, USA, 1996.
- [15] G. Licea, J. Favela, Design patterns for the development of synchronous collaborative systems, Proceedings of the International Workshop on Software Technology, Mexico, DF, November 1998.
- [16] T. Malone, K. Lai, C. Fry, Experiments with oval: a radically tailorable tool for cooperative work, Proceedings of the CSCW'92, November, 1992.
- [17] S. Nurcan, Analysis and design of co-operative work process: a framework, Information and Software Technology 40 (3) (1998) 143–156.
- [18] M. Pendergast, S. Hayne, Groupware and social networks: will life ever be the same again?, Information and Software Technology 41 (6) (1999) 311–318.
- [19] A. Prakash, H. Shim, Dist view: support for building efficient collaborative applications using replicated objects, Proceedings of CSCW'94, Chapel Hill, NC, USA, 1994.
- [20] N. Pryce, Abstract session: an object structural pattern, Proceedings of EuroPlop'97, Kloster Irsee, Germany, 1997.
- [21] M. Roseman, S. Greenberg, Building groupware with groupKit, in: M. Harrison (Ed.), Tcl/Tk Tools, O'Reilly Press, 1997, pp. 535–564.
- [22] A. Sane, The elements of pattern style, Proceedings of the Second Pattern Languages of Programs Conference, Addison-Wesley, Menlo Park, California, 1995.
- [23] D.C. Schmitdh, M. Fayad, R.E. Johnson, Software patterns, Communications of the ACM 39 (10) (1996) 36–39.
- [24] A. Schoenfeld, Domain specific patterns: conversions, persons and roles, and documents and roles, Proceedings of PloP'96, Allerton Park, Illinois, 1996.
- [25] C. Schuckmann, L. Kirchner, J. Schümmer, J. Haake, Designing object-oriented synchronous groupware with COAST, Proceedings of CSCW'96, Boston, USA, 1996.
- [26] S. Shirmohammadi, J. Oliveira, N. Georganas, Applet-based telecollaboration: a network-centric approach, IEEE Multimedia 5 (2) (1998) 64–73.
- [27] J. Simao, N. Pregaica, E. Domingos, J. Martins, Dagora: a flexible, scalable and reliable object-oriented groupware platform, Proceedings of the OOGP'97 Workshop in ESCW'97, Lancaster, UK, 1997.
- [28] J. Trevor, T. Koch, G. Woetzel, MetaWeb: bringing synchronous groupware to the world wide web, Proceedings of the European Conference on Computer Supported Cooperative Work, ECSCW'97, Lancaster, 1997.
- [29] M. Walther, Supporting development of synchronous collaboration tools on the web with GroCo, Proceedings of the ERCIM Workshop on CSCW and the Web, Sankt Augustin, Germany, February 1996.
- [30] M. Brown, M. Najork, Distributed active objects, Computer Networks and ISDN Systems 28 (7) (1996) 1037–1048.
- [31] F. Vitali, C. Chiu, M. Bieber, Extending HTML in a principled way with displets, Proceedings of the Sixth International World Wide Web Conference, California, USA, April 1997.
- [32] T. Rizzo, Programming Microsoft Outlook and Microsoft Exchange, 2nd ed., June, Microsoft Press, 2000.
- [33] [BSCW] Basic Support for Cooperative Work. <http://bscw.gmd.de>
- [34] [eRooms] Instinctive Technology, Inc. <http://www.instinctive.com>
- [35] [JAMM] Java Applets Made Multiuser. <http://simon.cs.vt.edu/JAMM>
- [36] [JETS] Java-Enabled Telecollaboration System. <http://www.mcrlab.uottawa.ca/jets>
- [37] [JSDT] Java Shared Data Toolkit. <http://www.javasoft.com/products/javamedia/jsdt/index.html>
- [38] [NetMeeting] Microsoft NetMeeting. <http://www.microsoft.com>
- [39] [Netopia] Netopia Virtual Office. <http://www.netopia.com/software/nvo/win/overview.html>
- [40] [Notes] Lotus Notes. <http://www.lotus.com/home.nsf/tabs/lotusnotes>
- [41] [NSTP] Notification Service Transfer Protocol. <http://nstp.research.lotus.com>
- [42] [TeamWave] TeamWave Software Ltd. <http://www.teamwave.com>