Design Patterns for Collaborative Systems

Luis A. Guerrero
Depto. de Ciencias de la Computación
Universidad de Chile
luguerre@dcc.uchile.cl

David A. Fuller
Depto. de Computación
Pontificia Universidad Católica de Chile
dfuller@ing.puc.cl

Abstract

Collaborative applications provide a group of users with the facility to communicate and share data in a coordinate way. In this paper we propose a pattern system to design the 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.

1. Introduction

A *design pattern* [7] 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, 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 [18] a pattern paper tells the story of a design decision: in some *context*, if one faces a *problem* where the *solution* must satisfy certain requirements, or *forces*, then one must use the *solution*; but keep in mind that certain *implementation* issues will be faced.

Individual patterns can be interwoven within *patterns 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 how to avoid costly traps and pitfalls which have been traditionally learned through experience [19].

This paper describes a pattern system for collaborative applications design. Section 2 describes the studied tools and applications, and elaborates on the nine patterns of the pattern system. Section 3 explains the relationships among the patterns when we design the collaborative applications. Finally section 4 presents our conclusions.

2. Design Patterns in Collaborative Applications

In deliberate and focusing on the identification of design patterns for collaborative applications, certain tools were studied for the construction of these applications. These tools are: GroupKit [17, GroupKit], NSCA Habanero [5, Habanero], JSDT ("Java Shared Data Toolkit") [JSDT], NSTP ("Notification Service Transfer Protocol") [6, NSTP], GroCo ("Group Communications") [25], MetaWeb [24], COAST ("COoperative Application System Toolkit") [21], CBE ("Collaboratory Builder's Environment") [12], Artefact [3], Mushroom [10], DistView [15], Oval [14], JETS ("Java-Enabled Telecollaboration System") [22] and DAgora [23].

Moreover, we studied a number of collaborative applications which have demonstrated a certain degree of success. These include: Microsoft *NetMeeting* [NetMeeting], GMD *BSCW* ("Basic Support for Cooperative Work") [BSCW], Instinctive Technology Inc. *eRooms* [eRooms], Netopia Inc. *Netopia Virtual Office* [Netopia], TeamWave Software Ltd. *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. These concepts are: sessions, users, roles, events, objects, repositories, messages, views and broadcast. The *messages* and *events* concepts can be specified within the *broadcast* concept. Therefore, we can say that candidates to design patterns include: *sessions*, *users*, *roles*, *broadcast*, *objects*, *repositories,* and *views*. In addition to these seven common features of the toolkits for the construction of collaborative applications, two more can be included: *environments* and *floor control*. Utilizing these nine components, plus a *broker* pattern [Bush96], we can construct a prototype pattern-based framework for

the development of collaborative applications named TOP ("Ten Objects Platform") [8].

These above-mentioned components are explained in the following sections and are presented as patterns for the design and development of collaborative applications.

2.1. Pattern 1: Session (or workgroup)

Context

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.

Problem

How do we manage and coordinate workgroups in collaborative applications?

Forces
- Users of the collaborative applications work in groups.
- 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.

Solution

Maintain a list of all users belonging to the workgroup, as well as a list of users currently working (connected) at the moment. Request the name and password of each user requesting to work with the shared data and validate it.

Implementation

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 check the status of specific sessions. Protocol should exist requesting users to identify themselves as either a member or coordinator of a session, as well as to know if a user is currently connected (working). A protocol should exist for a user to enter a session, and to leave it.

Examples

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.

Related Patterns

The *abstract session* pattern [16] allows a server object to maintain each client's status it serves.

2.2. Pattern 2: Repository

Context

In general, all collaborative applications need to store and share data.

Problem

How to provide users with the necessary tools that will furnish them with the ability to consistently retrieve objects that have been created and saved by other collaborative application users? How can users share data?

Forces
- Users produce data as a result of collaborative work. Thus, they need to store that data.
- The possibility exists for all the users within the group to share produced data.
- A user can recover, remove or modify an object that belongs to the group.
- Sometimes it is necessary to organize the information in several data repositories.
- Certain applications need to deny access to data among specific users or during specified time periods.

Solution

Provide information repositories that can be managed from the applications. Furnish the necessary methods to manage the data objects contained within these repositories, in order for them to be shared among the users.

Implementation

Each repository should have information identified by its name, owner or creator, description, and state (whether the information is available or if it is restricted). A protocol should exist to store an instance of the repository, to check if a repository exists, to offer a list of a session repositories, to check if an object is available within the repository, to close the repository, to open it, and to erase it. Also, 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.

Examples

In some *brainstorming* systems a workgroup presents a grouping of ideas or proposals. Once the presentation phase of ideas ends, a voting in 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 like BSCW and eRooms.

Related Patterns

The *shared repository* pattern [11] is an architectural pattern that defines a communication model for software components based on shared memory.

2.3. Pattern 3: Object (or meta-object, or information object)

Context

In order to provide data awareness it is necessary, in most applications, to maintain information on each produced object as the user creates it, the creation date and time, the object type, some keywords to aid in searches, and previous versions of the objects.

Problem

The group work process generates data that needs to be managed. How do we manage information about this data?

Forces
- During interaction the users of the group create new objects or modify existing ones.
- In order to provide data awareness mechanisms each object should maintain information about the user who creates or modifies it.
- If we want to implement a system with multiple versions (for data awareness) it is necessary to store copies of the erased or modified objects, as well as pertinent information, of the user who erases or modifies it, as well as the moment (date and time) it is made.
- In order to satisfy object searches it is necessary to store keywords from each object (e.g. keywords that identify a specific image or a sound).

Solution

Provide an *information object* for each data object. This object or meta-object should contain information about the object's creation date and time, it's owner or creator, the object type, previous modifications, keywords, and previous versions.

Implementation

The *meta-object* 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 store an instance of an object, to check if an object exists, to recover the real name of an object, and to know an object type.

Examples

If we want to make queries on objects such as video, images or sounds 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, when it was last modified or when it was done (data awareness).

When sending "message" type objects to other users it is necessary that the receiving user can identify the time 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.

2.4. Pattern 4: View

Context

In many collaborative applications it is necessary to graphically display a repository's data. Various kinds of users can have different perspectives of the same data. Therefore, data repository has several views.

Problem

All users of the group should not necessarily be able to see the same thing when they want to see the data. How can we provide views of the repository data?

Forces
- All users need to see the data they generate, whether as individual objects or as a whole that integrates all the objects generated by all the users.
- Some data needs to be hidden from certain users.
- Some users require seeing data in a different way (bigger, in another order, etc.).
- Users should be able to choose to see the same data in another format (another view).

Solution

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.

Implementation

The *view* object type should have information with its name, its owner or creator, the repository (or repositories) it belongs to, the view type (for example an HTML page, an ordered list, or a SQL query), its description, the date and time of it's 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).

Examples

In a chat application users receive messages in the same order they were sent (orderly view). A course lecturer may see and modify the grades of all the students whom they are responsible for, but individual students see only their own grades without the ability to modify them. Systems such as COAST and Mushroom utilize the idea of *views*.

Related Patterns

The *observer* pattern [7], also called *publisher-subscriber* [4], defines a one-to-many dependence among objects in such a way that when an object changes all their dependencies are notified and updated automatically.

2.5. Pattern 5: Floor control

Context

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.

Problem

How to provide floor control on objects? What assignment policy can be applied that decides which control is granted to what user?

Forces
- 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 it is marked "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.

Solution

A *floor control* object type is associated to 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.

Implementation

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

Examples

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 to speak.

Related Patterns

Guillermo Licea and Jesús Favela describe in [13] a similar floor control object detailing the different assignment policies in passing such a control.

2.6. Pattern 6: Broadcast

Context

It is common that during a work session users need to send objects 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 (RPC), or notifications to other users or even to other applications.

Problem

How to send objects to other users within the workgroup?

Forces
- Sometimes users require sending objects to other users within the workgroup.
- Sometimes users require sending objects to users connected only at that moment.
- Sometimes users require sending objects to all the users within the group (connected and/or not connected).
- Sometimes the system requires sending events, commands, or notifications to all connected users.
- Sometimes the system needs to send events or notifications to all users, connected and not connected.

Solution

A broadcast technique takes charge of sending objects to users.

Implementation

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

Examples

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 "broadcast" idea is present in systems such as GroupKit, Habanero, MetaWeb and COAST.

Related Patterns

The *publisher-subscriber* pattern [4], also called *observer* [7], maintains a synchronized state of cooperating components. A publisher object notifies a number of subscriber's objects about any change in its state.

2.7. Pattern 7: User

Context

In a collaborative application several users make up a workgroup. It is necessary to maintain certain personal information for each user.

Problem

How to manage the information of the workgroup's users?

Forces
- When a user enters into an application it is necessary to check their name and password.
- To provide users' awareness, it is necessary to maintain certain information about each user so other group members may identify them.

Solution

A *user* object has all the necessary personal information for each of the users participating within the application.

Implementation

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

Examples

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".

2.8. Pattern 8: Role

Context

In many collaborative applications users have different access rights to information.

Problem

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

Forces
- Most collaborative applications have different types of users (roles).
- Certain data are only available for specified users.
- Certain operations (erase, modify, recover, see) or processes (backup, print, etc.) are restricted from certain users.
- Some data views are restricted from certain users.

Solution

Assign *roles* to the users which define each user's role: their access rights over the objects of a repository, and views.

Implementation

The *role* object maintains information with its name, the session to which it belongs, 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 kind of rights which apply them to specified repository objects in a finer granularity outline.

The role should implement methods to store an instance, to erase an instance, to check if a role exists, to check if an 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.

Examples

Within most collaborative text editors can be found two types of roles, 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 session's *coordinator*. Different user's roles can be defined in GroupKit and CBE.

Related Patterns

The *role object* pattern [2] defines an object which can change its role dynamically. The *persons and roles* pattern

[20] allows users to participate inside a system with different roles which they can change dynamically. The *documents and roles* pattern [20] shows the relationships between documents and people's roles within a system.

2.9. Pattern 9: Environment

Context

One collaborative application can have many work sessions. Each session may have different users working in separated 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.

Problem

Several workgroups may work within the same application without interfering with one another. How to provide uniform access to collaborative applications, identifying users among corresponding sessions?

Forces
- 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.
- All users among different sessions should have a uniform way to enter into a collaborative application. The system should be able to identify which session individual users may work.

Solution

The *environment* object has information on all work sessions or different users' groups which are using the collaborative application but working among various repositories.

Implementation

The environment maintains information with its name, description, owner or creator, and work sessions list (each session maintains its respective repositories list). The environment has methods to store an instance, to check if an environment exists, to erase an environment, and to return a list with all the environments of the system. When a user enters into an environment the environment checks and enters the user into the corresponding session.

Examples

Several user groups may utilize the same chat application. Other groups do not see the messages of a particular group.

Related Patterns

The *mediator* pattern [7] defines an object which encapsulates the way in which a group of objects interact.

3. A Pattern System for Collaborative Applications Design

The previous nine patterns define a *pattern system* for the design of collaborative applications. The figure 1 describes the way in which these patterns are related.

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
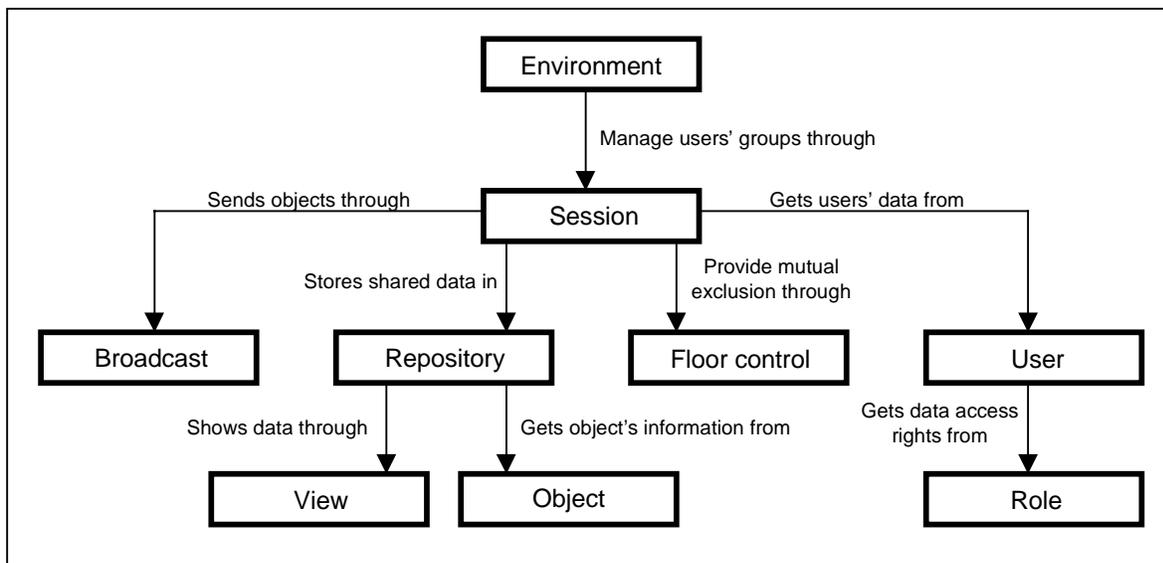

Figure 1. *The pattern system of TOP*

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

4. 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*, *users*, *roles*, *broadcast*, *floor control*, *repositories*, *views* 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 for several user groups simultaneously sharing an application without interfering with others groups. This allows for transforming *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 new sessions for other groups to use the applications.

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

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 may 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 the collaborative systems development easier.

5. References

[1] Agerbo, E. and Cornils, A. "How to Preserve the Benefits of Design Patterns". Proceedings of OPSLA'98, Vancouver, Germany, 1998.

[2] Bäumer, D., Riehle, D. Siberski, W. and Wulf, M. "The Role Object Pattern". Proceedings of the PLOP'97, Monticello, Illinois, USA, 1997.

[3] Brandenburg, J. et al. "Artefact: A Framework for Low-Overhead Web-Based Collaborative Systems". Proceedings of CSCW'98, Seattle, Washington, 1998.

[4] Buschmann, F. Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. "Pattern-Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996.

[5] Chavert, A., Grossman, E., Jackson L., Pietrowics, S. and Seguin, C. "Java Object-Sharing in Habanero". Communications of the ACM, Vol. 41, No. 6, pp. 69-76, June, 1998.

[6] Day, M., Patterson, J. and Mitchell, D. "The Notification Service Transfer Protocol (NSTP): Infrastructure for Synchronous Groupware", Sixth International World Wide Web Conference, Stanford, California, USA, 1997.

[7] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1994.

[8] Guerrero, L. and Fuller, D. "Objects for Fast Prototyping of Collaborative Applications". Proceedings of 4th CYTED-RITOS International Workshop on Groupware, CRIWG'98, Rio de Janeiro, Brazil, September, 1998.

[9] Johnson, R. "Documenting Frameworks using Patterns". Proceedings of the OOPLA'92, pp. 63-76, Vancouver, Canada, 1992.

[10] Kindberg, T. "Mushroom, A Framework for Collaboration and Interaction across the Internet". Proceedings of the ERCIM Workshop on CSCW and the Web, Sankt Augustin, Germany, February, 1996.

[11] Lalanda, P. "Shared Repository Pattern". Proceedings of PLOP'98, Monticello, Illinois, USA, 1998.

[12] Lee, J., Prakash, A., Jaeger, T. and Wu, G. "Supporting Multi-User, Multi-Applet Workspaces in CBE". Proceedings of CSCW'96, Boston, Massachusetts, USA, 1996.

[13] Licea, G. and Favela, J. "Design patterns for the development of synchronous collaborative systems". Taller Internacional de Tecnología de Software, Simposium Internacional de Computación CIC-98 Mexico D.F., November, 1998.

[14] Malone, T., Lai, K. and Fry, C. "Experiments with Oval: A Radically Tailorable Tool for Cooperative Work". Proceedings of the CSCW'92, November, 1992.

[15] Prakash, A., and Shim, H. "DistView: Support for Building Efficient Collaborative Applications using Replicated Objects". Proceedings of CSCW'94, Chapel Hill, NC, USA, 1994.

[16] Pryce, N. "Abstract Session: An Object Structural Pattern". Proceedings of EuroPlop'97, Kloster Irsee, Germany, 1997.

[17] Roseman, M. and Greenberg, S. "Building Groupware with GroupKit". In M. Harrison (Ed.) Tcl/Tk Tools, pp. 535-564, O'Reilly Press, 1997.

[18] Sane, A. "The Elements of Pattern Style". Proceedings of the Second Pattern Languages of Programs Conference, Addison-Wesley, Menlo Park, California, 1995.

[19] Schmitdh, D. C., Fayad, M., and Johnson, R. E. "Software Patterns". Communications of the ACM, Vol. 39, No. 10, pp. 36-39, October, 1996.

[20] Schoenfeld, A. "Domain Specific Patterns: Conversions, Persons and Roles, and Documents and Roles". Proceedings of PloP'96, Allerton Park, Illinois, 1996.

[21] Schuckmann, C., Kirchner, L., Schümmer, J. and Haake, J. "Designing object-oriented synchronous groupware with COAST". Proceedings of CSCW'96, Boston, USA, 1996.

[22] Shirmohammadi, S., Oliveira, J., and Georganas, N. "Applet-Based Telecollaboration: A Network-Centric Approach". IEEE Multimedia, Vol. 5, No. 2, pp. 64-73, 1998.

[23] Simao, J., Preguica, N., Domingos, E. and Martins, J. "Dagora: A Flexible, Scalable and Reliable Object-Oriented Groupware Platform". Proceedings of the OOGP'97 workshop in ESCW'97, Lancaster, UK, 1997.

[24] Trevor, J., Koch, T. and Woetzel, G. "MetaWeb: Bringing Synchronous Groupware to the World Wide Web". Proceedings of the European Conference on Computer Supported Cooperative Work, ECSCW'97, Lancaster, 1997.

[25] Walther, M. "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.

URL References

[BSCW] Basic Support for Cooperative Work.
http://bscw.gmd.de

[eRooms] Instinctive Technology, Inc.
http://www.instinctive.com

[GroupKit] GroupKit.
http://www.cpsc.ucalgary.ca/projects/grouplab/projects/groupkit

[Habanero] NCSA Habanero.
http://www.ncsa.uiuc.edu/SDG/Software/Habanero

[JSDT] Java Shared Data Toolkit.
http://www.javasoft.com/products/javamedia/jsdt/index.html

[NetMeeting] Microsoft NetMeeting.
http://www.microsoft.com

[Netopia] Netopia Virtual Office.
http://www.netopia.com/software/nvo/win/overview.html

[Notes] Lotus Notes.
http://www.lotus.com/home.nsf/tabs/lotusnotes

[NSTP] Notification Service Transfer Protocol.
http://nstp.research.lotus.com

[TeamWave] TeamWave Software Ltd.
http://www.teamwave.com

Acknowledgments