An Architectural Model for Component Groupware

Cléver R.G. de Farias[1,2], Carlos E. Gonçalves[2], Marta C. Rosatelli[2], Luís Ferreira Pires[3], and Marten van Sinderen[3]

[1] Departamento de Física e Matemática,
Faculdade de Filosofia Ciências e Letras de Ribeirão Preto (FFCLRP/USP),
Av. Bandeirantes, 3900, 14040-901 – Ribeirão Preto (SP), Brazil
farias@ffclrp.usp.br
[2] Programa de Mestrado em Informática, Universidade Católica de Santos,
Rua Dr. Carvalho de Mendonça, 144, 11070-906 – Santos (SP), Brazil
{cleverfarias, ceg-elus, rosatelli}@unisantos.edu.br
[3] Centre for Telematics and Information Technology, University of Twente,
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{pires, sinderen}@cs.utwente.nl

Abstract. This paper proposes an architectural model to facilitate the design of component-based groupware systems. This architectural model has been defined based on (1) three pre-defined component types, (2) a refinement strategy that relies on these component types, (3) the identification of layers of collaboration concerns, and (4) rules for the coupling and distribution of the components that implement these concerns. Our architectural model is beneficial for controlling the complexity of the development process, since it gives concrete guidance on the concerns to be considered and decomposition disciplines to be applied in each development step. The paper illustrates the application of this architectural model with an example of an electronic voting system.

1 Introduction

The technological advances of the last decade have brought many changes into our society. Computers have become essential working and entertainment tools. Yet, most of the computer systems are targeted to single users, although most of our working tasks are likely to involve a group of people. Systems that provide support for groups of people engaged in a common task are called *groupware systems*.

The development of groupware systems poses many different challenges. Apart from the social aspects of groupware, developers are faced with problems typical of both distributed systems and cooperative work. Problems pertaining to distributed systems are, amongst others, the need for adequate levels of transparency, reliability, security, and heterogeneity support. Problems related to cooperative work are mainly the need for flexibility, integration, and tailorability in groupware systems [5].

The use of component-based technologies contributes to solve these problems [2, 8, 11, 19, 21, 22]. Component-based development aims at constructing software artefacts by assembling prefabricated, configurable and independently evolving building blocks called components. A component is a binary piece of software, self-contained, customisable and composable, with well-defined interfaces and dependencies.

H. Fuks, S. Lukosch, and A.C. Salgado (Eds.): CRIWG 2005, LNCS 3706, pp. 105 – 120, 2005.

Components are deployed on top of distributed platforms, contributing to solve many of the distribution-related problems of groupware systems. Components can also be configured, replaced and combined on-the-fly, which enhances the degree of flexibility, integration and tailorability provided by a system.

One of the biggest challenges in system development is the definition of the system architecture. The architecture of a (computing) system can be defined as the structure (or structures) of the system in terms of software components, the externally visible parts of those components and the relationships among them [3]. In this way, the architecture can be seen as the top-level decomposition of a system into major components, together with a characterisation of how these components interact [23].

A proper definition of the architecture of a system facilitates not only the system design as a whole but also the development and reuse of components for a family of similar systems, the so-called product line development. Thus, our work proposes an architectural model to help groupware developers tackling the design of component-based groupware systems. Our architectural model defines different types of components that serve as basis for system and component refinements. Our model also defines different types of collaboration concerns to help the identification of the different types of components and the assignment of functionality to components.

The remainder of this work is structured as follows: section 2 discusses the refinement strategy adopted in this work; section 3 identifies component types to be applied in the (component-based) development process of groupware systems; section 4 proposes a set of consecutive layers, one for each specific collaboration aspect of a cooperative work process; section 5 introduces the concept of collaboration coupling between these layers and discusses some related distribution aspects; section 6 illustrates the application of our architectural model with a case study related to an electronic voting system; finally, section 7 presents some conclusions.

2 Refinement Strategy

In the design of a groupware system, we use of the concept of functional entity as an abstraction for an entity in the real world (e.g., a system, a system user or a system component) capable of executing behavior. A functional entity executes behavior by itself or in cooperation with other functional entities, which form the environment of this entity.

2.1 Refinement Principle

There are two main approaches to tackle the refinement of a functional entity in general: (1) to refine the interactions between the functional entity and its environment without changing the granularity of the functional entity itself, i.e., without decomposing the functional entity into smaller parts, or (2) to decompose the functional entity into smaller parts and allocate the functional entity interactions to these parts without changing these interactions, except for the introduction of new (internal) interactions between the smaller parts. The first approach is called *interaction refinement*, while the second one is called *entity refinement* [16].

Fig. 1 illustrates the difference between the interaction refinement and entity refinement in the refinement of a system into system parts. Fig. 1 also shows that these approaches can be combined in successive refinement steps to produce some design, in which both the system is decomposed into smaller parts and the interactions are refined into more detailed interactions.

In the context of this work, we consider that a refinement process is carried out only according to the entity refinement approach. Consequently, we assume that the interactions between a functional entity and its environment are preserved as we refine the system into a set of interrelated components. Therefore, unless explicitly mentioned, we use the term *refinement* or *decomposition* to denote entity refinement.

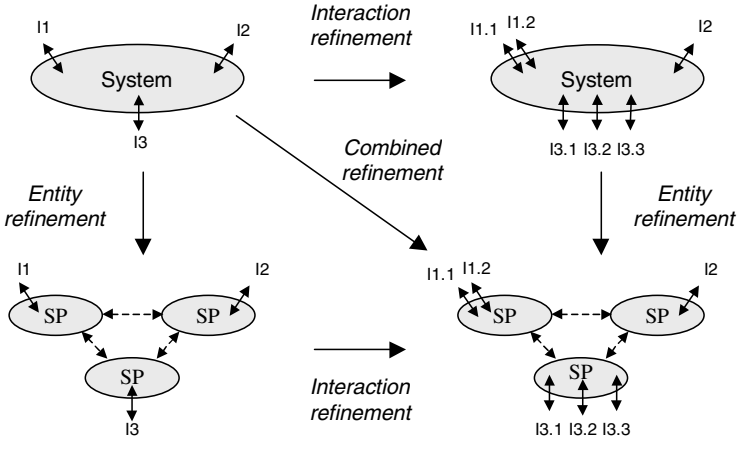

Fig. 1. Alternative refinement approaches

2.2 Component Decomposition

We can identify two slightly different approaches regarding the decomposition of a system into components: the *continuous recursion* approach [4] and the *discrete recursion* approach [7].

In the continuous recursion approach, the system is continuously refined into finer-grained components, until components of a desired granularity level or complexity are identified. Since in this case no specific component types are defined beforehand, this approach can only provide general guidelines for reducing the complexity of a component.

In the discrete recursion approach, the system is systematically refined into components of different types, which are pre-defined according to, for examples, different objectives or milestones identified throughout the design trajectory. A component type defines a number of characteristics common to a number of components. Different component types can be made to correspond to different component granularities, although this is not always necessarily the case. This decomposition approach is capable of providing both general and more specific refinement guidelines for each component type.

Fig. 2 illustrates the difference between the two approaches. Fig. 2a shows the continuous recursion refinement approach, in which all components defined in the consecutive decomposition steps are of the same type ('grey' components). Fig. 2b shows the discrete recursion refinement approach, in which two component types are defined beforehand (grey and white components). Furthermore, Fig. 2b shows a decomposition discipline in which only a single component type (either grey or white) is used at a certain level of granularity.

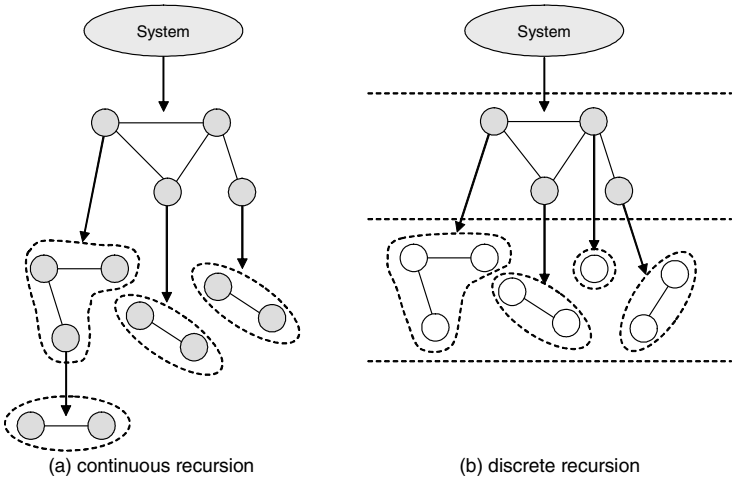

Fig. 2. Alternative component decomposition approaches

Fig. 2 also shows that these approaches are not fundamentally different (it is all about successive decomposition), and that discrete recursion could even be seen as a specialisation of continuous recursion.

3 Component Types

In this work we use the discrete recursion approach for component decomposition, because this approach allows us to tailor the component types according to different (sets of) concerns. Thus, we identify three different types of components inspired by [7]: *basic components*, *groupware components*, and *application components*.

A basic component is the most basic unit of design, implementation, and deployment. A basic component is not further refined into other components. Additionally, the behaviour of a basic component is carried out by binary code. Therefore, an instance of a basic component runs on a single machine, which is usually part of a distributed environment.

A groupware component consists of a set of basic components that cooperate in order to provide a mostly self-contained set of groupware functions. A groupware component embodies the behaviour corresponding to an independent collaborative concept or feature that can be reused to build larger groupware components and systems.

The self-containment of a groupware component does not imply that this component is isolated from other components. On the contrary, a groupware component should be composable, i.e., one should be able to compose different groupware components, and these components should be able to interact with each other. However, such a component should have minimal dependencies in order to maximize its reuse.

A groupware component also encapsulates distribution aspects. Since a groupware component consists internally of basic components, and basic components can be distributed individually across a network, the distribution aspects normally required by a groupware component are consequently addressed by the composition of (distributed) basic components. However, basic components can be used to address not only the physical distribution aspects, but also the distribution of concerns and responsibilities that form a groupware component.

An application component corresponds to a groupware application, i.e., an independent application that can be used separately or integrated into another groupware system. Any groupware system under development can be considered an example of an application component. However, groupware components can also be used as building blocks for larger application components. In most cases, an application component consists of a set of interrelated groupware components that cooperate in order to provide some application-level functionality.

In order to illustrate this component hierarchy, we take a videoconferencing system as an example. This system can be seen as a composition of individual applications, such as videoconferencing, chat, and shared whiteboard applications. A videoconferencing application can be decomposed into separate groupware components, which provide, e.g., audio support, video support, and attendance support. An audio support component can be decomposed into separate basic components to handle the connection establishment, coding, decoding, transmission, and so on.

Nevertheless, this classification scheme is flexible and subject to the designer's choice and interpretation. For example, in the videoconferencing system above, one could alternatively assign a separate groupware component to handle each of the videoconferencing, chat, and shared whiteboard concerns. In this alternative, the system is seen as a composition of individual groupware components, instead of a composition of application components.

4 Collaboration Concerns

In order to structure groupware systems we have identified layers of collaboration concerns that have to be handled by these systems. We have also identified rules for configuring these layers so that a meaningful groupware system can be obtained.

4.1 Collaboration Concern Layers

Suppose a particular groupware component manages the editing of a shared document. This component is responsible for maintaining the consistency of the document, allowing multiple users to change the document simultaneously. Initially, a user may choose to have a different view of the document. For example, a user may choose an "outline" view, as opposed to another user who uses a "normal" view at the same

time. The question is whether this particular choice of the first user should affect the way in which the second user views the document or the component should allow different users to have different views of the document simultaneously.

Consider that this particular component is also responsible for keeping the users informed about changes in the document. Another question is whether a user should be notified of every change in the document or any action of another user, or the component should only notify the user when a change affects the part of the document this specific user is currently working on.

These are typical issues that have to be dealt with by a groupware component. In order to provide flexibility to deal with these and other issues, we identify a number of so-called *collaboration concern layers*, on which different aspects of the functionality of a groupware component can be positioned.

We have identified four separate layers: *interface*, *user*, *collaboration*, and *resource*. Each layer uses the functionality provided by the layer below in order to provide some functionality that is used by the layer above. Fig. 3 depicts the collaboration concern layers identified in this work and their relationships.

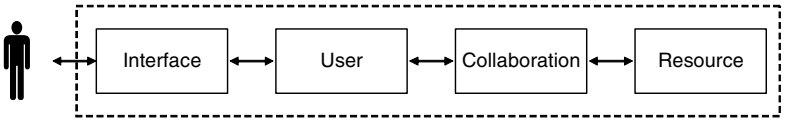

Fig. 3. Collaboration concern layers

The interface layer is concerned with providing a suitable interface between a human user and the groupware component. Considering a drawing component of a shared whiteboard application, the interface layer should enable a user to create new drawings and change or delete existing drawings by means of graphical commands. The interface layer should also enable the user to visualize the drawing itself through the interpretation of drawing commands. Therefore, the interface layer handles all the direct communication with the users via a graphical user interface.

The user layer is concerned with the local support for the activities performed by a single user. The user layer addresses the local issues with respect to each individual user that do not affect the collaboration as a whole. For example, suppose that our drawing component enables a user to make changes to a local copy of a shared drawing, without changing the shared drawing immediately. This allows the user to incorporate changes to the shared drawing only when this user is absolutely satisfied with these changes. Therefore, the user layer maintains a user's perception of the collaboration. The user layer also supports the interface layer, by relating it with the collaboration layer.

The collaboration layer is concerned with collaboration issues of multiple users. The logic involved in different collaboration aspects, such as communication, coordination and cooperation functionalities [6, 9], are mainly tackled at this layer. Considering our drawing component, the collaboration layer should be able to handle the drawing contributions of multiple users, relating them as necessary. Therefore, this layer is responsible for the implementation of the core aspects of the collaboration and for relating the user layer to the resource layer.

The resource layer is concerned with the access to shared collaboration information (resources), which could be, for example, kept persistently in a database. In our drawing component, the resource layer should be able to store the drawing and drawing commands, saving and loading them as necessary. The resource layer is only accessible through the collaboration layer.

4.2 Implementation of Concern Layers

Each collaboration concern layer can be implemented by one or more basic components. An interface component implements the interface collaboration layer. Similarly, user, collaboration, and resource components implement the user, the collaboration, and the resource collaboration layers, respectively. Nevertheless, it is not uncommon to someone implement more than one layer using a single component, in case the functionality provided by these layers is simple enough to be implemented by a single component.

We distinguish between three different types of functionality interfaces that a component can support based on the purpose and visibility (scope) of the interface: *graphical user interfaces*, *internal interfaces* and *external interfaces*.

A graphical user interface (GUI) supports the interactions between a human user and an interface component. An internal interface supports the interactions between the components of a single groupware component. Such an interface has internal visibility with respect to a groupware component, i.e., a groupware component cannot interact with another groupware component using internal interfaces. An external interface supports the interactions between groupware components. Such an interface has external visibility with respect to groupware components.

Interface components and resource components usually do not have external interfaces. Therefore, interactions between groupware components are normally only achieved via the user and collaboration components.

A groupware component does not need to have all four layers. For example, it is only meaningful to have a resource layer if some shared information has to be stored persistently. Similarly, an interface layer is only meaningful if the component interacts with a human user.

Nevertheless, if a groupware component has more than one layer, these layers should be strictly hierarchically related. For example, the interface layer should not access the collaboration layer or the resource layer directly, nor should the user layer access the resource layer directly. Thus, a single groupware component should consist of at least one and up to four ordered collaboration concern layers.

Fig. 4 illustrates some examples of groupware components that conform to the rules given above. Each layer is represented by a corresponding basic component. A groupware component is represented by a dashed rectangle, while a basic component is represented as a solid rectangle. An interface is represented by a T-bar attached to a component, while an operation invocation is represented by an arrow leading to an interface. An external interface is represented by a solid T-bar that crosses the boundary of the groupware component, while an internal interface is represented by a dashed T-bar inside the groupware component. Graphical interfaces are not explicitly represented.

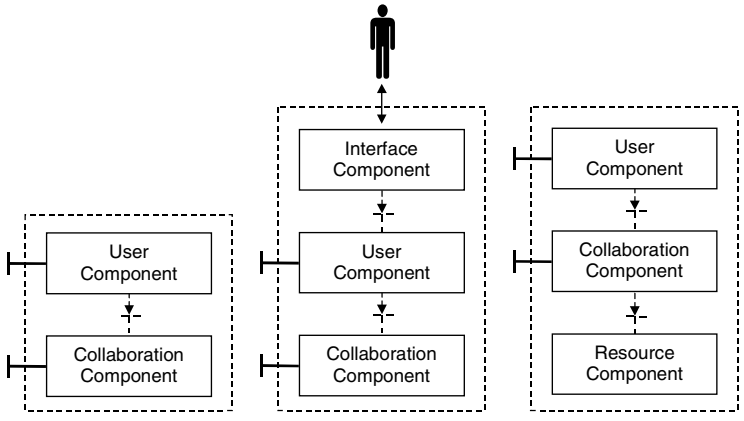

Fig. 4. Valid layering distributions

4.3 Related Work

Our collaboration concern layers have been identified based on a number of developments, such as Patterson's state levels [15], Herzum and Sims' distribution tiers [7] and Wilson's architectural layers [24]. These developments address similar issues although with different terminology.

Patterson [15] identifies four state levels in which a synchronous groupware application can be structured, namely display, view, model, and file. The display state contains the information that drives the user display. The view state contains the information that relates the user display to the underlying information in the application, which is the model state. The file state consists of a persistent representation of the application information. Based on Patterson's state levels, Ter Hofte proposes a four-level collaborative architecture known as the zipper architecture [22].

Herzum and Sims [7] propose a distribution architecture for business components that consists of four tiers, namely user, workspace, enterprise, and resource. These tiers are roughly equivalent to our collaboration concern layers. However, this architecture emphasizes distribution aspects, instead of the collaboration aspects that we emphasize in our work.

Table 1. Collaboration concern layers and related approaches

Collaboration Concern Layers	Patterson's State Levels	Herzum and Sims' Distribution Tiers	Wilson's Architectural Layers
Interface	Display	User	View
User	View	Workspace	Application-Model
Collaboration	Model	Enterprise	Domain
Resource	File	Resource	Persistence

Wilson [24] proposes the development of a distributed application according to four architectural layers, namely view, application-model, domain and persistence. The view layer deals with user interface issues, the application-model layer deals with application-specific logic, the domain layer deals with domain-specific logic, and the persistence layer deals with the storage of information in a persistent format.

Table 1 shows the correspondence between our collaboration concern layers, Patterson's state levels, Herzum and Sims distribution tiers, and Wilson's architectural layers.

5 Collaboration Coupling

The coupling of collaboration concerns and their logical or physical distribution are two important aspects to be considered in the design of groupware systems.

5.1 Coupling Levels

So far we have discussed the collaboration concern layers of a groupware component in the context of a single user. However, the whole purpose of groupware systems is to support interactions involving multiple users.

In the context of a groupware system, and particularly in the context of a groupware component, two or more users may or may not share the same perception of the ongoing collaboration supported by the system. For example, in the case of the groupware component that manages the editing of a shared document, if one user chooses an outline view of the document instead of a normal view, this decision could possibly affect another user's view of the document. In case all the component users share the same perception, the outline view would replace the normal view for all users. Otherwise, the other users do not share the same perception of the collaboration, and only that particular user would have an outline view of the document.

We can apply the same reasoning to each collaboration concern layer of a groupware component. As a consequence, collaboration concern layers can be coupled or uncoupled. *Coupling* was introduced as a general mechanism for uniting the interaction contributions of different users, such that users might share the same view or state of a collaboration [22].

A collaboration concern layer of a groupware component is coupled if all users have the same perception of the information present in the layer and how this information changes. Therefore, a collaboration concern layer across multiple users can be coupled or uncoupled. An important property of coupling is downwards transitivity, which means that if a layer is coupled, the layers below, from the interface layer down to the resource layer, must be coupled as well in order to ensure consistency.

Four levels of coupling can be established based on the collaboration layers defined in this work: *interface*, *user*, *collaboration*, and *resource* coupling.

The interface coupling level represents the tightest coupling level. All the component users have the same perception of the collaboration, starting at the user interface layer. This level corresponds to the collaboration style known as What You See Is What I See (WYSIWIS) [20].

The user coupling level offers more freedom (independence of use) to the component user than the interface coupling level. All the component users have the same perception of the collaboration starting at the user layer, i.e., the information at the user layer is shared by all users, but their interface layers are kept separate.

The collaboration coupling level goes a step further and offers more freedom than the user coupling level. All the component users have the same perception of the collaboration starting at the collaboration layer, i.e., the information at the collaboration layer is shared by all users, but their interface and user layers are kept separate.

The resource coupling level offers the loosest coupling level. All component users have the same perception of the collaboration only at the resource layer, i.e., the information at the resource layer is shared by all users, but their interface, user and collaboration layers are kept separate.

Fig. 5 depicts the collaboration coupling levels defined in this work for two users. A large rectangle labelled with the layer initial indicates that the layer it represents is coupled, while a small rectangle also labelled with the layer initial indicates that the layer it represents is uncoupled.

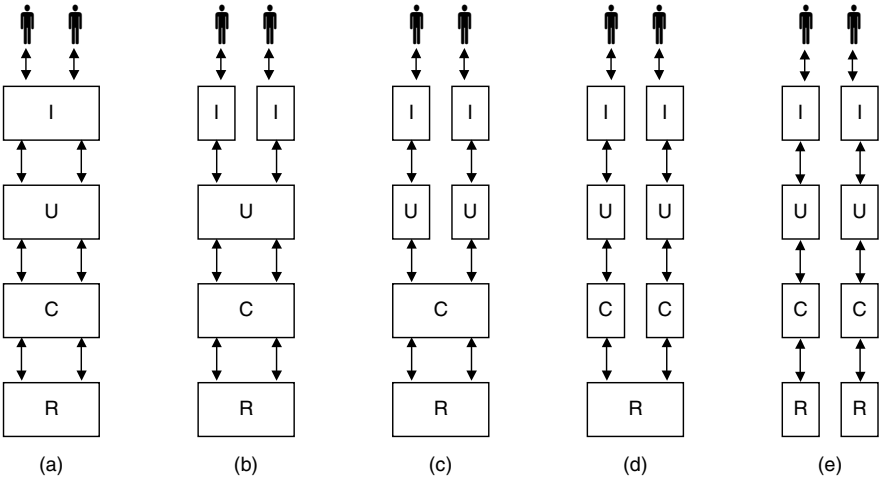

Fig. 5. Collaboration coupling levels

Fig. 5a to Fig. 5d show the interface, user, collaboration and resource coupling levels, respectively. Fig. 5e depicts the absence of coupling at all levels, i.e., in this figure the two instances of the groupware component operate independently from each other (offline collaboration).

In a truly flexible groupware system, the users of this system should be able to choose between the different levels of coupling, changing it at run-time based on the characteristics and requirements of the task at hand. They should also be able to choose a temporary absence of coupling, i.e., the users may decide to work independently for a while,

resuming their coupling status sometime later. In this case, additional mechanisms to guarantee the consistency of the collaboration afterwards have to be implemented.

5.2 Distribution Issues

There are basically two ways to achieve collaboration coupling in a given layer: using a centralised architecture or using a replicated architecture with synchronization mechanisms.

In a centralized architecture, a single copy of the collaboration state, i.e., all the information contained in a layer, is maintained and shared by the users of the layer. Concurrency control mechanisms should be used to avoid inconsistencies if needed.

In a replicated architecture, multiple copies of the collaboration state are maintained, one for each user of the layer. In this case, synchronization mechanisms (protocols) are used to maintain the consistency across the replicated copies of the collaboration.

The collaboration state of an uncoupled layer is non-centralised by definition, i.e., each user maintain its own copy of the collaboration state. However, the collaboration state of a given coupled layer can be either centralised or replicated. Thus, different architectures for each coupling level can be applied in a single groupware system.

The resource layer can only be coupled otherwise we are actually talking about distinct instances of a collaboration in place of a single one (see example in Fig. 5e). This implies that the resource coupling level can only be centralised or replicated. Each layer above can be coupled or uncoupled; in case the layer is coupled, the coupling level can be again centralised or replicated. A fully centralised architecture has only centralised layers, a fully replicated architecture has only replicated layers, and a hybrid architecture combines centralised and replicated layers. However, once a coupled layer is implemented using a centralised architecture, all layers below should also be implemented using a centralised architecture. This is because it makes little sense to centralise some information in order to assure consistency, and then to replicate some other information upon which the first one depends on.

Although possible in principle, it is very unlikely that interface coupling is achieved using a centralized architecture because of the complexity involved and response time requirements. Interface coupling is usually implemented based on shared window systems (see [1, 10, 20]).

Fig. 6 illustrates three possible combinations of centralised and replicated architectures to achieve coupling at the collaboration layer for two users. In Fig. 6 a rectangle represents an instance of a basic component, which is labelled with the initial of the layer it implements. A large rectangle indicates a component in a centralized architecture, while two small rectangles connected by a synchronization bar (a double-edged horizontal arrow) indicate a component in a replicated architecture. Small rectangles without a synchronization bar indicates that the layer they represent are uncoupled. Fig. 6a depicts a fully centralised architecture (both the resource and the collaboration layers are centralised), Fig. 6b depicts a hybrid architecture (the resource layer is centralised while the coordination layer is replicated), and Fig. 6c depicts a fully replicated architecture (both the resource and the collaboration layers are replicated).

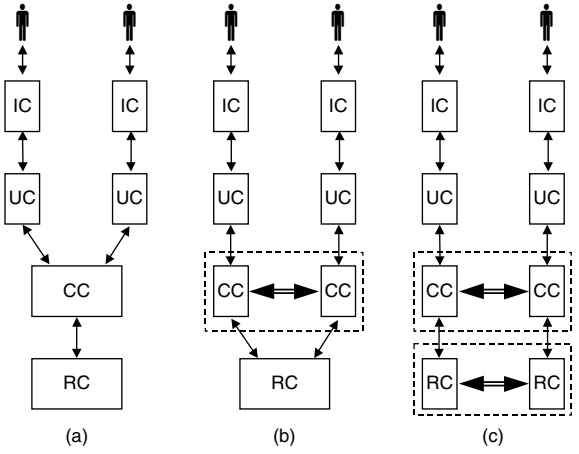

Fig. 6. Centralised, hybrid, and replicated architectures

The choice between a centralised architecture and a replicated architecture is mainly related to implementation issues [22]. A centralised architecture is indicated whenever a given coupling level either requires some specialised or excessive processing power that prevents replication or changes from a coupled state to an uncoupled state at this level are unlikely. A replicated architecture is indicated whenever changes from a coupled state to an uncoupled state are desired, thus improving the flexibility of the component.

A discussion on the benefits and drawbacks of centralised versus replicated architectures, as well as the mechanisms used to achieve consistency in both architectures, falls outside the scope of this work. For detailed discussions on these issues we refer to [1, 10, 14, 17, 18, 22].

6 Design of an Electronic Voting System

In order to exemplify our architectural model we have applied it in the development an Electronic Voting System (EVS).

The EVS basically enables its users to create polls and/or vote on them. To use the EVS, any user is required to register first. Registered users can then log in and out of the system and update their personal profile. Any registered user can open a poll, defining its subject, voting options and eligible participants. Once a poll is registered, it will be available for voting to all selected participants. A registered user can visualize a list of all the polls available in the system and cast a single vote to any open poll in which she participates. Additionally, the results of a closed poll should be available for consultation by all users.

The design of the EVS system was carried out in a number of design steps according to the guidelines provided in [5]. In this work, we used UML 2.0 [12, 13] as our modelling language. Fig. 7 depicts the high level architectural model of the EVS using UML component diagram.

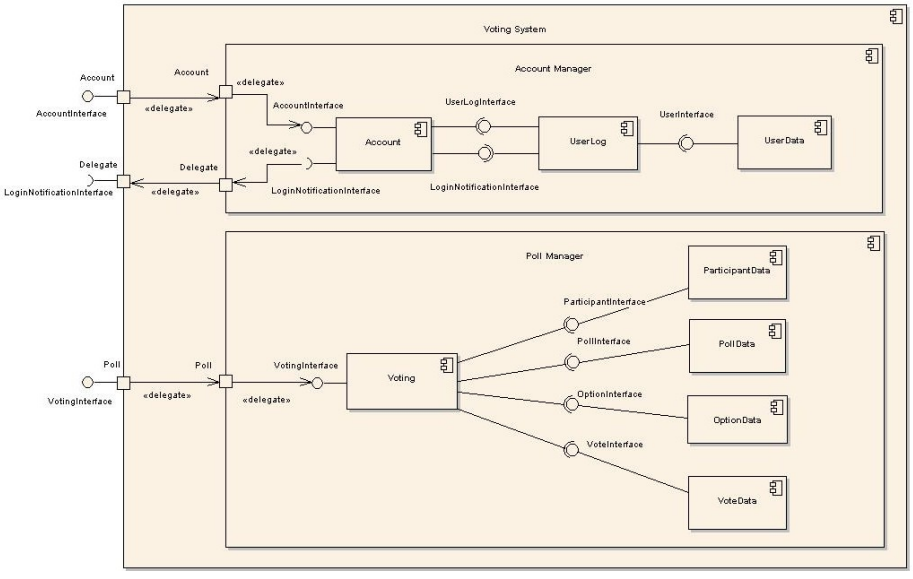

Fig. 7. EVS high level architecture

In the first step the EVS service was specified. At this point the EVS was seen as an application component, called Voting System. In the second step, this component was decomposed into two groupware components:

- Account Manager, which is responsible for the registration of users, their logging on and off the system, and provision of user awareness;
- Poll Manager, which is responsible for the control over the creation of polls, as well as the cast of votes;

In the third step, each groupware component was then refined into a number of simple components. In this case study we did not consider the interface layer as part of the groupware component, but considered it as part of the client application.

The Account Manager component was refined into the basic components Account, UserLog and UserData, which implement the user, collaboration and resource layers respectively. The identified components are coupled at the collaboration coupling level, using a fully centralised architecture for the coupled layers. Since the user layer is uncoupled, a separate instance of the component Account should be created to support each separate user of this component.

The Poll Manager component was refined into the basic components Voting, ParticipantData, PollData, OptionData and VoteData. The component Voting implements the user and collaboration layers, while the remaining components implement the resource layer. The identified components are coupled at the resource coupling level, using a fully centralised architecture for this layer. Since the user/collaboration layer is uncoupled, a separate instance of the component Voting should be created to support each separate user of this component.

The EVS components were implemented as Enterprise Java Beans (EJB) components. As an implementation infrastructure, we used the JBOSS 3.2 application server, the Hypersonic database, and JAAS for authentication. The client was developed using SWING and a communication library of the JBOSS server.

The components Account and Voting were implemented as two stateless session beans, while the component UserLog was implemented as a statefull session bean. The remaining components were implemented entity beans, using bean managed Persistence. Notification across components was implemented using the Java Message Service (JMS).

Fig. 8 depicts some screenshots of the voting system user interface. Fig. 8a shows the system main interface. This interface shows the list of current logged users. Fig. 8b shows the poll registration interface. Fig. 8c shows the poll voting interface. Fig. 8d shows the poll results interface.

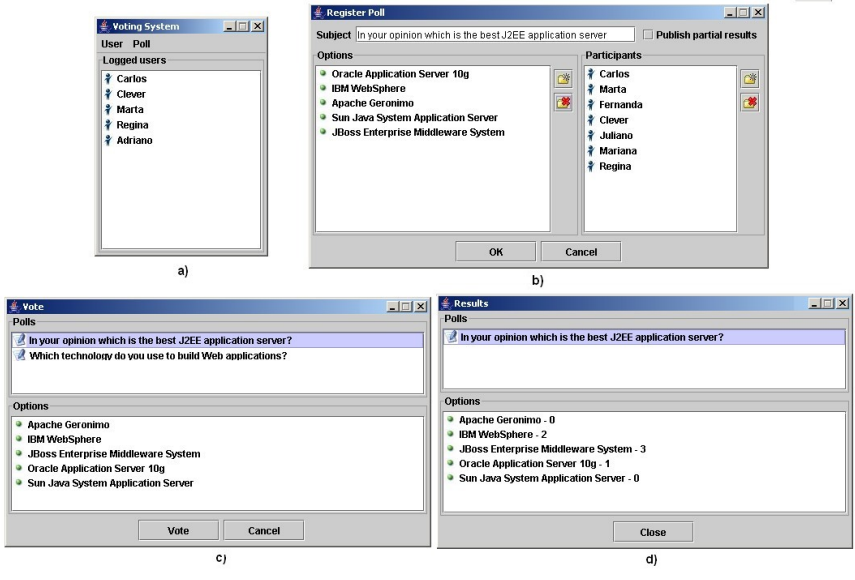

Fig. 8. Voting system user interface

7 Conclusion

This paper proposes an architectural model for the development of component-based groupware systems. According to this model, a groupware system is recursively decomposed into a number of interrelated components, such that the service provided by the groupware system is provided by the composed individual services of these components.

The decomposition process is carried out according to a discrete recursion approach based on pre-defined component types. We believe that the use of specific component types facilitates the decomposition process if compared to a decomposition approach

that does not make such distinction. Additionally, the use of pre-defined component types facilitates the identification and reuse of existing components.

We have discussed the use of collaboration concern layers and collaboration coupling to structure the different collaboration aspects within the scope of a groupware component. We believe that the use of these layers and guidelines facilitates the logical and physical distribution of these aspects and the assignment of functionality to components thereafter thus improving reuse and speeding up the development process.

We have illustrated the application of the architectural model proposed in this work by means of a simple case study describing the development of an electronic voting system, which is being developed in the scope of the project TIDIA-Ae[1].

Acknowledgements

This work has been partially supported by FAPESP under project number 2003/08279-2.

References

1. Ahuja, S.R., Ensor, J.R. and Lucco, S.E.: A comparison of application sharing mechanisms in real-time desktop conferencing systems. In *Proceedings of the 1990 ACM Conference on Office Information Systems (COIS'90)*, pp. 238-248, 1990.
2. Banavar, G., Doddapaneti, S., Miller, K. and Mukherjee, B.: Rapidly Building Synchronous Collaborative Applications by Direct Manipulation. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (CSCW'98)*, pp. 139-148, 1998.
3. Bass, L., Clements, P. and Kazman, R.: *Software Architecture in Practice*. Addison-Wesley, 1997.
4. D'Souza, D. F. and Wills, A. C.: *Objects, Components and Frameworks with UML: the Catalysis Approach*. Addison Wesley, 1999.
5. de Farias, C. R. G.: *Architectural Design of Groupware Systems: a Component-Based Approach*. PhD Thesis, University of Twente, the Netherlands, 2002.
6. Fuks, H., Raposo, A. B., Gerosa, M. A. and Lucena, C. J. P.: Applying the 3C Model to Groupware Development. In *International Journal of Cooperative Information Systems (IJCIS)*, 14(2-3), pp. 299-328, 2005.
7. Herzum, P. and Sims, O.: *Business component factory: a comprehensive overview of component-based development for the enterprise*. John Wiley & Sons, 2000.
8. Hummes, J. and Merialdo, B.: Design of Extensible Component-Based Groupware. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 53-74, 2000.
9. Laurillau, Y. and Nigay, L.: Clover Architecture for Groupware. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work (CSCW'02)*, pp. 236-246, 2002.
10. Lauwers, J.C., Joseph, T.A., Lantz, K.A. and Romanow, A.L.: Replicated architectures for shared window systems: a critique. In P*roceedings of the 1990 ACM Conference on Office Information Systems (COIS'90)*, pp. 249-260, 1990.

[1] http://tidia-ae.incubadora.fapesp.br/novo/

11. Litiu, R. and Prakash, A.: Developing adaptive groupware applications using a mobile component framework. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW'00)*, pp. 107-116, 2000.
12. OMG: *UML 2.0 Infrastructure Specification*. Adopted Specification, Object Management Group, 2003.
13. OMG: *UML 2.0 Superstructure Specification*. Revised Final Adopted Specification, Object Management Group, 2004.
14. Patterson, J.F., Day, M. and Kucan, J.: Notification servers for synchronous groupware. In *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, pp. 122-129, 1996.
15. Patterson, J.F.: A taxonomy of architectures for synchronous groupware applications. *SIGOIS Bulletin*, 15 (3), pp. 27-29, 1995.
16. Quartel, D., Ferreira Pires, L. and Sinderen, M.: On Architectural Support for Behaviour Refinement in Distributed Systems Design. In *Journal of Integrated Design and Process Science*, 6 (1), pp. 1-30, 2002.
17. Roth, J. and Unger, C.: An extensible classification model for distribution architectures of synchronous groupware. In *Designing Cooperative Systems: the Use of Theories and Models, Proceedings of the 5th International Conference on the Design of Cooperative Systems (COOP'00)*, pp. 113-127, 2000.
18. Schuckmann, C., Kirchner, L., Schümmer, J. and Haake, J.M.: Designing object-oriented synchronous groupware with COAST, In *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, pp. 30-38, 1996.
19. Slagter, R. J.: *Dynamic Groupware Services: Modular design of tailorable groupware*. PhD Thesis, University of Twente, the Netherlands, 2004.
20. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S. and Tatar, D.: WYSIWIS revised: early experiences with multiuser interfaces. ACM Transactions on Office Information Systems, 5(2), pp. 147-167, 1987.
21. Teege, G.: Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 101-122, 2000.
22. ter Hofte, G. H.: *Working Apart Together: Foundations for component groupware*. PhD Thesis, Telematics Institute, the Netherlands, 1998.
23. van Vliet, H.: *Software Engineering: Principles and Practice*. John Wiley & Sons, USA, 2000.
24. Wilson, C.: Application Architectures with Enterprise JavaBeans. *Component Strategies*, 2(2), pp. 25-34, 1999.