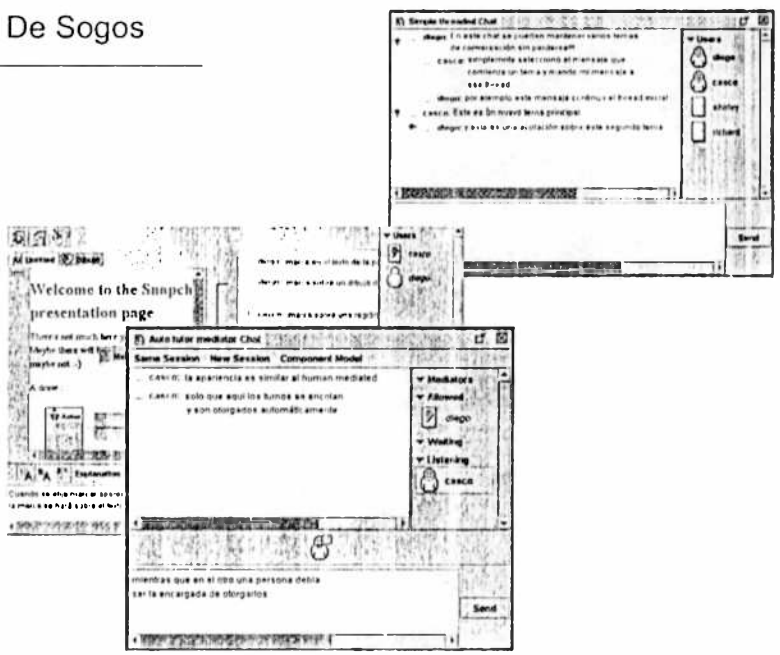
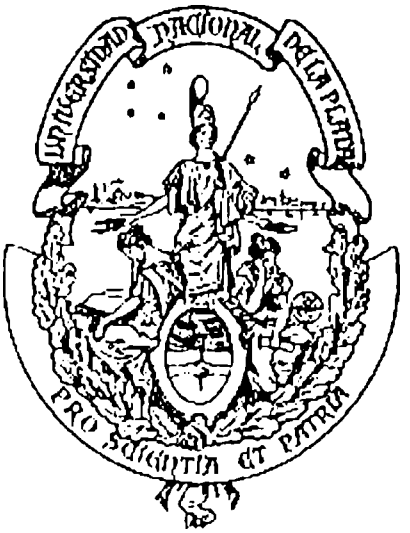
Chatblocks

Un framework para desarrollo de aplicaciones para la comunicación basada en texto

Diego De Sogos



Universidad Nacional de La Plata



Chatblocks

Un framework para el desarrollo de aplicaciones para la comunicación basada en texto

Tesis de Grado de
Licenciatura en Informática

Facultad de Informática,
Universidad Nacional de La Plata

Disertante
Diego De Sogos

Director: Dr. Gustavo Rossi
Co-Director: Lic. Alejandro Fernández

Fecha de Presentación: 26 de marzo de 2002
La Plata, Pcia. de Buenos Aires
Argentina

Agradecimientos

Este trabajo de grado marca el final de una carrera que llevó varios años de aprendizaje y experiencia. Cualquier ciclo como el marcado por esta tesis, se vuelve siempre una montaña muy difícil de escalar sin el apoyo y colaboración de un conjunto de personas excelente que siempre esta dispuesto a brindar una mano a quien la necesite. A continuación pasaré a nombrar a alguna de ellas:
A Federico Naso quien me tendió una mano amiga que siempre me tuvo presente en todo y me dio la oportunidad de poder participar en el LIFIA como un nuevo integrante de su grupo.
A Fernando García y Ricardo Tesoriero, personas sin cuyos aportes este trabajo no hubiera sido posible. Nando inicio todo este proyecto con la formulación de su prototipo del framework y fue junto a Richard dos excelentes personas tanto en el trabajo como fuera de él.
A Alejandro Fernández (Casco), por ser el hombre que más incentivó mis ideas y proyectos, que confió siempre en todo nuestro grupo de trabajo y por su colaboración y consejo en este proyecto.
Y por supuesto hago extensivo este saludo para todos los lifianos que me permitieron ser uno mas del grupo.
A todos ellos mi mas sincero agradecimiento.

La Plata, marzo de 2002

Contenido

Capítulo 1. Introducción

En la actualidad, la comunicación entre los integrantes de un proyecto de trabajo, compañeros de estudios, o simplemente amigos, puede llevarse a cabo prácticamente bajo cualquier circunstancia, ya no es necesario estar enfrentados cara a cara para mantener una discusión, ni siquiera es necesario de disponer de un teléfono, las grandes redes globales de comunicación nos mantienen en contacto en cualquier lugar del mundo.
Dentro de las herramientas de nueva generación, sin dudas la que más utilidad esta brindando es el correo electrónico. Sin embargo, no es una herramienta óptima cuando lo que se quiere es hacer una consulta que requiera respuesta inmediata por parte del interlocutor, la característica inherentemente asincrónica del correo electrónico contribuye una dificultad, en este tipo de comunicación, en donde por ahora el rey sigue siendo el teléfono.

Pero para un grupo de personas que requieren una comunicación rápida y fluida, sin necesidad de tener que cambiar de entorno de trabajo, con la posibilidad de compartir de forma inmediata el trabajo que estén realizando, estas herramientas convencionales no son suficientes. Es en este punto es que entran en escena las nuevas tecnologías informáticas ofreciendo toda una nueva gama de aplicaciones destinadas a facilitar el trabajo grupal, proveyendo al usuario de un espacio de comunicación y cooperación entre los miembros de su grupo. A estas tecnologías se las encuadra en el rubro de CSCW (Computer Supported Collaborative Work), es decir el área del groupware (me remitiré a una definición mas precisa de estos dos términos mas adelante), que emplea la computadora como el elemento básico sobre el que se apoyan las herramientas para el trabajo en grupo.

Volcándonos exclusivamente a la tarea de ofrecer dentro de lo que es CSCW una herramienta flexible, ágil, fácil de utilizar, que no requiera de equipamiento (hardware) adicional, nos encontramos con los sistemas de "chat".
Un sistema es una aplicación multiusuario, en donde dos o más personas pueden intercambiar mensajes (en su modalidad más simple, solo de texto) de forma sincrónica, en tiempo real.

Este tipo de herramientas surgió casi con la misma aparición del correo electrónico, y con el correr de los años han ido ganando mas popularidad, sobre todo al surgir aplicaciones

con interfaces más intuitivas y fáciles de utilizar por el común de la gente. Sin embargo, aunque su aspecto se modificó bastante, aún siguen manteniendo la idea original, simplemente servir como un canal de comunicación sincrónico entre dos o más personas, sin ningún tipo de control o regulación acerca del carácter y momento en que los usuarios pueden realizar sus aportes.

Si pensamos sobre todo en el área de la educación, estas herramientas presentan unas serias dificultades, entre las que podes mencionar: 1. la falta de organización la cual queda a cargo de los integrantes u organizadores de la reunión virtual, 2. la incapacidad de enriquecer la comunicación agregando algún otro tipo de semántica adicional al mensaje de texto (en chats tradicionales, una forma que tienen de agregar mayor poder de expresión en los mensajes es dándole al usuario la posibilidad de, a través de ciertos caracteres combinados, mostrar una figura que represente algún estado de ánimo o emoción), y 3. poca o ningún tipo de personalización de los sistemas para cambiar o decidir que política o reglas se quiere aplicar para la discusión, es más, en su mayoría las reglas son fijadas en forma implícita entre los participantes, cuando uno pregunta algo, por ejemplo, sabe que debe esperar un mensaje de parte del resto antes de volver a preguntar, como es de esperarse en una conversación cara a cara.

1.1 Motivación

Atacando las falencias mencionadas en la sección anterior, surgió la idea de construir Chatblocks, como un framework que le brinda al desarrollador de aplicaciones del tipo descrito, una forma rápida y totalmente extensible y personalizable de construir sistemas chat no convencionales, mas adelante explicaré con mas precisión a que me refiero con "no convencionales".

El por qué un framework y no una aplicación concreta sobreviene de forma inmediata si pensamos que lo que se pretende es un sistema en donde el usuario este totalmente libre para diseñar y construir el protocolo de comunicación que desee, sin atarse a ninguna restricción. Esta libertad en el desarrollo del protocolo debe estar acompañada por el resto de las partes del sistema, que deberán adaptarse y configurarse de acuerdo a dicho protocolo para componer de esta forma un sistema completo. De esta forma un framework presenta el esqueleto básico y provee los ganchos donde deberán montarse las partes concretas. El resultado es un sistema chat capaz de atender a cualquier tipo de necesidad o característica del dominio.

Particularmente, este trabajo de grado fue de especial interés para el Instituto de Investigación y Desarrollo Fraunhofer, cuya división IPSI-Concert, ubicada en la localidad de Darmstadt, Alemania, cuenta con proyectos de investigación que estudian específicamente la mejor forma de trabajar en grupo en aulas virtuales, y en el área de enseñanza de lenguajes empleando para ello CSCW. El interés de este Instituto radicaba en querer tener un conjunto de herramientas chats que permitieran simular diferentes ambientes de discusión como pueden ser debates, dar una clase, o ser el orador de una conferencia.

Como idea original surge un trabajo de tesis previo a éste que sembró las semillas de cómo debía ser un framework que soporte este tipo de aplicaciones, incluso llegó a presentarse un diseño prototípico del mismo realizado en Smalltalk. Con esta tesis se pretende la construcción de una nueva versión completa y totalmente funcional basada en ese diseño original, pero con la inclusión de otros factores que se detallan en la sección siguiente.

1.2 Objetivo

El propósito de este trabajo de grado es proveer de una herramienta que sirva para el desarrollo de sistemas chat, sobre todo para la experimentación académica en el aprendizaje humano, y como mecanismo para proveer aplicaciones flexibles, totalmente adaptables y capaza de ser incluidas como herramientas básicas en sistemas groupware de mayor envergadura.
Para alcanzar esta meta, es necesario la construcción de un framework, que soporte el desarrollo y deployment de Componentes Groupware. En secciones subsecuentes aclararé a que me refiero con este término.

Una primera aproximación a este framework fue realizada en un trabajo previo de tesis de grado elaborado por el Lic. Fernando García. En su tesis, García ofrece un diseño de este framework, Chatblocks, pensado para un ambiente cooperativo proporcionado por COAST, un framework implementado en Smalltalk, que permitía al desarrollador independizarse del mecanismo de transporte y distribución de la información compartida por la red física, y crear aplicaciones colaborativas de forma sencilla siguiendo unas cuantas reglas o restricciones con respecto a la implementación.
Estos prototipos mostraban las características principales que se persiguen con la idea de aportar un framework para sistemas chat.

Quedaron ciertos aspectos no definidos o inconclusos en el diseño del prototipo. Entre ellos puedo mencionar la carencia de componentes cerradas de software que permitan ver una aplicación construida con Chatblocks como si se tratara de una conjunción de componentes, de forma similar a como un desarrollador Java elabora nuevas componentes usando Swing.

En esta tesis, pretendo presentar una implementación final, funcional y utilizable de dicho framework, pero realizada en una plataforma totalmente distinta y apoyado esta vez en otro framework denominado DyCE que permite la generación y carga automática de lo que en DyCE se llaman Componentes Groupware. Como principal diferencia este nuevo ambiente esta totalmente construido y basado en la tecnología Java, y sobresale por encima de COAST por proveer al usuario de un servidor HTPP, que hace posible que las componentes creadas con DyCE puedan ser desarrolladas y luego distribuidas a través del Web, por parte de cualquier desarrollador.

Entre los objetivos que se persiguen con esta versión de Chatblocks sobreviene un conjunto de modificaciones al diseño original, no solo para adaptarlo a DyCE, sino

también para extender su funcionalidad original con otro tipo de operaciones como puede ser el manejo de contribuciones clasificadas por tipo (otra forma de aumentar la expresividad y de enriquecer el protocolo), o un manejo de login que ayude al administrador a del sistema a llevar un control o una futura planificación del uso de la aplicación. Además permite cumplir con una de las metas principales de Chatblocks que es la posibilidad de brindar al desarrollador componentes de software reutilizable

A todo esto también debo agregar la experiencia adicional que generó el emplear DyCE y las posibles extensiones que planteo sobre el mismo, las cuales surgieron como necesidad durante el desarrollo de Chatblocks.

Finalmente la tesis proveerá también un conjunto de aplicaciones finales realizadas íntegramente empleando el framework Chatblocks y que sirven como prueba de cuánta utilidad puede brindar el mismo. Estas aplicaciones surgen a pedido de IPSI-Fraunhofer como parte de su proyecto de enseñanza

Podría resumir los objetivos que pretende cubrir esta tesis en:

- Rediseñar, adaptar, y extender el framework Chatblocks, para abarcar aspectos no tenidos en cuenta en su versión original como la generación y distribución de componentes reutilizable de software.
- Ofrecer una implementación final de Chatblocks, realizada en Java
- Presentar un análisis de en que consiste el framework DyCE, y ofrecer una serie de herramientas (clases que se podrían ver como una extensión al framework) que facilitan la tarea de implementar componentes groupware
- Implementar un conjunto de sistemas chat utilizando el framework que para ese fin desarrollamos. De este modo el desarrollador adquiere una mejor noción de lo que puede obtener con la utilización de Chatblocks

1.3 Organización de la Tesis

Como he explicado en la sección anterior, la tesis encara un objetivo principal que es la implementación de Chatblocks, y como consecuencia de la misma una serie de objetivos secundarios que deben ser tenidos en cuenta.
A tal fin el resto de este trabajo se estructuró de la siguiente forma.

El capítulo 2, se especifican los puntos clave a tener en cuenta en una aplicación final desarrollada con Chatblocks, así como también las facilidades que debe proveer el mismo al desarrollador de la aplicación. Se provee un panorama de cuales son los aspectos nuevos a tener en cuenta en esta versión final del framework.

En carácter de introducción y como forma de presentar el punto de partida para este trabajo, se ofrece en el Capítulo 3, el diseño y la arquitectura general del framework pensada originalmente para ser empleado con el framework COAST. En dicho capítulo se presenta la filosofía del COAST, y se especifica la arquitectura general del prototipo de Chatblocks.

En el capítulo siguiente, se presentan los conceptos fundamentales para entender en que consisten los Componentes Groupware, base de toda aplicación o desarrollo hecho con DyCE. Además se ofrece una pequeña introducción acerca de en que consiste un framework orientado a objetos.

El Capítulo 5 desarrolla todo lo referente a la filosofía del framework DyCE, como permite la generación de componentes y las ideas principales ligadas a como se debe realizar la construcción de una componente groupware. El manejo de sesiones y la arquitectura cliente-servidor que DyCE plantea son mencionado sin dar demasiados detalles de implementación, solo lo necesario como para que el lector pueda comprender como es el desarrollo en DyCE, y en que forma fue pensado y diseñado Chatblocks.

Los conceptos centrales de la tesis son presentados en el capítulo 6, allí se presentan con detalles cada una de las partes que componen Chatblocks, mostrando la arquitectura total del sistema, y ofreciendo ejemplos sencillos acerca de cómo se construyen aplicaciones empleando este framework. Para facilitar su compresión y lectura he divido al capítulo en dos partes: filosofía del framework y utilización del mismo.

Se han incluido en un capitulo aparte, el 7, todo lo referente a extensiones o creación de nuevas herramientas para el DyCE, como son las destinadas a facilitar el empleo de componentes estándares de Java Swing, sin necesidad de tener que subclasificar o realizar algún otro tipo de modificaciones en las clases del modelo compartido de una componente DyCE.

Todas las conclusiones y temas que surgieron como nuevas puntas de investigación son expuestas en el Capítulo 8.

Para dar una muestra del tipo de aplicaciones que se pueden construir con Chatblocks, se introduce el Anexo A, en el cuál se presenta a una familia de sistemas chat, similares en apariencia pero de funcionamiento bastante diferente entre sí, sobre todo por los diferentes protocolos de comunicación con que fueron ensamblados. El Snapchat es un exponente mas que sobresaliente acerca de la versatilidad obtenida con el Chatblocks.

Por último, resta mencionar el Anexo B, que contiene los diagramas del diseño general del framework y el Anexo C conteniendo información curricular del autor de esta tesis.

Capítulo 2. Requerimientos del Framework

El framework debe permitir la creación y configuración de aplicaciones con funcionalidad y características lo suficientemente variadas como para cubrir el amplio número de diferentes formas de comunicación existentes entre los humanos. Es deseable poder utilizar el framework para armar aplicaciones que vayan tanto desde simular un debate de política como una clase en el ámbito educativo.

Otro punto importante que debe cubrir el framework, es el de brindar un atajo al desarrollador, ofreciéndole una estructura básica, pero a la vez muy flexible, que le deslinde todo tipo de responsabilidad acerca de pensar como hará para lograr enviar un mensaje de una máquina a otra, o de cómo hará para especificar diferentes tipos de usuarios, como registrar el ingreso o salida de los mismos, etc.

2.1 Requerimientos del Usuario final

Podemos enumerar las siguientes características en un sistema chat, cada aplicación en particular debería poder optar por alguna de las opciones brindadas para cada característica:

R1. Numero de Participantes:
Cantidad de usuarios que participan de la conversación.
En general, pueden clasificarse dos grandes grupos de sistemas chat de acuerdo a esta característica:
Chat de dos y no más que dos participantes. En donde la conversación se puede mantener prácticamente sin inconvenientes a través de un protocolo implícito de comunicación
Chats que permiten mas de dos participantes
Con mas de dos personas, aún manteniendo un único tema de conversación, el aporte de cada persona y el momento en que se realizan se vuelven confusos. Es aquí donde cobra

aún mayor importancia alguna ayuda adicional de la herramienta que permita a los usuarios una mejor organización de sus contribuciones

R2. Tipo de Comunicación:
Este requerimiento apunta al grado de obligatoriedad de la presencia online de los participantes, de acuerdo a si es necesaria o no, podemos definir dos tipos de sistemas chat:

R2.1 Sincrónicos: Significa que todos los participantes deben estar conectados (online) para poder participar de la conversación, cuando un participante abandona la sala de chat (es decir pasa a un estado offline), ya no es posible continuar enviándole mensajes.

R2.2 Asincrónicos: No impone la necesidad de la conexión, de cada participante, para llevar a cabo la conversación. En este caso puede pensarse en el chat como una casilla de correo central donde cada usuario decide entrar en algún momento, ver los mensajes depositados por el resto, y en caso de desearlo dejar alguno propio. No es habitual en los chats este tipo de comportamiento, generalmente se utilizan listas de discusión de correo electrónico para este tipo de situaciones.

Puede ser útil si el sistema aloja ambos tipos de comunicación, de esta forma, en caso que en una reunión, falte uno de los integrantes, el resto puede comenzar igual la discusión, y en todo caso permitir la incorporación tardía del usuario retrasado (el cuál debería tener la posibilidad de ver todas las contribuciones hechas por sus compañeros)

R3. Unidad de Envío de Información:
Generalmente los participantes pueden realizar contribuciones de dos formas básicas:

Caracter tipeado: Cada vez que el usuario presiona una tecla, el caracter ingresado es enviado a todos los receptores. En este tipo de sistemas, se genera un trafico continuo de información por la red, y muchas suele ser inútil, generalmente es más fácil de entender los mensajes cuando se pueden leer por completo y de corrido, en este caso los receptores ven todas las letras ingresadas de a una por vez, incluso las que el emisor borra casi al instante cuando se ha producido un error de tipeo.

Mensaje: El usuario escribe su contribución localmente y luego mediante una acción determinada lo envía a los receptores.

R4. Tipo de Almacenamiento:
El sistema chat puede mantener en forma persistente todas las contribuciones hechas en una discusión, de forma que los usuarios pueden acceder a este historial en futuras sesiones. En tal caso consideramos que el almacenamiento es permanente
Por el contrario, cuando los mensajes sólo son mantenidos por la aplicación durante el tiempo que dura la discusión, estamos ante un almacenamiento temporal. En estos casos es común incluir alguna utilidad en la aplicación que permita la exportación de todas las contribuciones a un archivo de texto.

Generalmente los chat accesibles vía web, poseen almacenamiento volátil. Mientras que muchos de los programas de mensajería instantánea (un caso de chat para dos participantes), suelen guardar un historial de las sesiones pero de forma local en cada máquina cliente. Esto nos lleva a otra clasificación en cuanto al tipo de almacenamiento:

Almacenamiento Distribuido, las sesiones se mantienen pero solo en cada máquina cliente. No es posible que se continúen después y sirven solo a modo de historial.

Almacenamiento Central, las sesiones se guardan en una base de datos centralizada, esto permite continuar fácilmente con una sesión determinada, o por Ej. La posibilidad de que un usuario que se incorpora a una sesión ya iniciada pueda leer todas las contribuciones hechas con anterioridad a su llegada de modo que puede ponerse al corriente de todo lo conversado

R5. Perfiles de Usuario:
La aplicación puede establecer o no, diferenciaciones entre los usuarios. De este modo pueden configurarse funcionalidades diferentes para cada participante de acuerdo al perfil de usuario que éste posea. Por ej. Puede existir un administrador, que sea el único capaza de invitar a otros usuarios a unirse a la sesión.
Otro usuario podría servir como moderador de la discusión, este tipo de funcionalidad que se diferencia de acuerdo al perfil del usuario, es poco habitual en los sistemas tradicionales. Sin embargo cobra vital importancia cuando se piensa en estas aplicaciones como herramientas para el estudio o aprendizaje.

R6. Protocolo de Comunicación:
Un concepto intrínsecamente relacionado con aplicaciones groupware es el de **floor control,** consiste en la imposición de ciertas reglas o restricciones a los usuarios para realizar ciertas acciones como en el caso de un sistema chat, la posibilidad de realizar una contribución. Generalmente la ocurrencia de una acción por parte de un participante, deviene en un cambio en el flujo de control de la aplicación, esta matriz de cambios conformada por la acción y el estado a los que conduce dicha acción, es lo que se denomina floor control.
¿En que sentido se puede encontrar floor control en un sistema chat?
El floor control esta dado por el protocolo existente para la discusión.
En los chat tradicionales, dicho protocolo es implícito y queda librado a la buena voluntad de los participantes que lo respeten. En estos casos desde el punto de vista de la aplicación podemos decir que no posee floor control.

R7. Elección del Receptor:
Dentro de un chat con mas de dos personas activas en la discusión es común la posibilidad de elegir a quienes se quiere hacer llegar el mensaje:
Broadcast: Cada mensaje es visible por todos los participantes de la conversación.
Privados: El emisor especifica a cual de los participantes esta destinado su mensaje.

Generalmente la segunda opción en los chat tradicionales solo permite indicar un único receptor para los mensajes privados.

R8. Semántica de las contribuciones:
Especifica el significado de un mensaje desde el punto de vista de la aplicación.
Cuando hablo de semántica, me refiero a proveer información adicional, complementaria al texto que conforma el cuerpo del mensaje.
Esta información adicional, puede ser simplemente inexistente como ocurre en los chats tradicionales, o como puede ser común encontrar en ciertos sistemas asincrónicos como las listas de correo, pueden contener referencias a mensajes enviados con anterioridad, aclarando de esta forma, cuales son los diferentes temas de discusión dentro la

conversación. Otro tipo de semántica adicional la puede proveer el adjuntar al mensaje alguna referencia a otro objeto como por ejemplo, que el mensaje sea también un link a una página web.

R9. Organización de la Información:
Las contribuciones recibidas pueden ser ordenadas, filtradas o visualizadas de diferente forma por cada usuario, de la misma forma que hacemos con nuestro lector de correo o nuestro cliente de news.

El sistema puede ordenar los mensajes de acuerdo a:
Atributos del mensaje: En este caso, el usuario podría optar por ordenar los mensajes por el orden de llegada (el timestamp del mensaje), lo cual es el orden habitual de un chat, o podría decidir ver los mensajes ordenados por autor del mismo

Semántica adicional: En este caso, puede tener en cuenta si el mensaje es una referencia a otros mensajes previos, o por el tipo de mensaje en si, por ejemplo si es un hipervínculo.

Es importante tener en cuenta que el usuario no debe perder de vista los mensajes nuevos que vayan arribando, es decir, que el hecho de aplicar un criterio de ordenación no conlleve a la pérdida de la información: si el usuario esta observando los mensajes ordenados por referencias a otros, entonces hay que decidir como se le hace notar que un mensaje nuevo a sido emitido si este no se encuentra dentro de las referencias a las que el usuario esta prestando atención, pero teniendo en cuenta que no se puede simplemente sacar al usuario abruptamente del lugar en que se encuentra para mostrarle el mensaje nuevo. Este tema de mensajes como referencias a otros, se lo conoce como Threaded Communication y constituye toda un área de investigación que abarca mas que nada a psicólogos y sociólogos, además de informáticos.

Además de aplicar un criterio de ordenación, puede ser posible tener diferentes formatos de visualización de los mensajes:
Por ejemplo, en los chat tradicionales, el formato más trivial y común es ver los mensajes emitidos en una campo de texto, generalmente con estructura de lista.
Algunos plantean un formato de historieta en donde cada mensaje nuevo conforma un nuevo cuadro de la historieta, en la que se ve un personaje representativo de la persona emisora.
En otros casos, puede ser útil una interfaz icónica que muestre los mensajes como notas que se van adhiriendo a una pizarra
En el caso de una threaded communication puede ser útil poseer una estructura de árbol en donde cada nodo raíz este conformado por el mensaje que inicia un nuevo hilo de la conversación

La combinación entre visualización y criterio de ordenación permite que la aplicación obtenga el grado de personalización requerida para diferentes tipos de usuarios y situaciones encontradas en las conversaciones.
Por ejemplo si ordenamos los mensajes por fecha de creación, podríamos mostrarlos en forma de lista. El usuario que desea ver los mensajes de un día determinado tendrá que recorrer todos los mensajes anteriores al mensaje deseado, esta situación no existiría si la

visualización fuese en forma de árbol cuyas ramas principales fuesen los días de creación de los mensajes.

R10. Awareness:

Es la información presente en la aplicación relacionada con la presencia de cada usuario dentro del sistema, debe permitir dar cierto grado de conocimiento acerca de que es lo que esta haciendo en todo momento cada participante

En una conversación virtual, podemos tener awareness vinculado con:

1. Participación: Saber quienes están participando de la charla
2. El tipo de conexión: para cada participante conocer en que estado se encuentra (offline, online, ocupado, inactivo, etc.)
3. E tipo de actividad: De cada participante, se conoce si esta escribiendo, dibujando, o si solo esta observando los mensajes emitidos.

El awareness es una de las funciones más importante que debe estar presente en un sistema cooperativo, ya que el hecho de que las personas que se están comunicando se hallan en diferentes lugares físicos, el proceso de interactuar entre ellos requiere cierto conocimiento de que se encuentra haciendo cada uno.

El awareness es un aspecto muy importante en todas las aplicaciones cooperativas, ya que es necesario que usuarios situados en diferentes lugares físicos, interactúen en la creación de algo, ya sea un concepto, una discusión o los planos de una nueva central hidroeléctrica. Para eso cada participante debe conocer quien esta recibiendo la información que el que transmite, y de esa forma emular las situaciones que se dan en la realidad.

El awareness de participación, intenta recrear la visualización de los usuarios, nótese que pueden existir usuarios que actúan como "oyentes", o sea que no realizan aportes de ningún tipo a la conversación pero sin embargo están participando de la conversación.

El awareness de conexión es muy importante en chats que soporten comunicación sincrónica, ya que permite identificar quienes están online permitiendo elevar el nivel de interacción.

El awareness de tipeo, tiene sentido en los chats sincrónicos, ya que esta indicando que el usuario esta preparando un mensaje para ser enviado, dejando a criterio de los demás participantes el esperar a que sea enviado dicho mensaje. Este podría verse como awareness de pàrticipación, ya que adelanta la participación de un usuario en la conversación.

R11. Post-Edición:

Consiste en brindar al usuario herramientas para la edición de la información ya enviada.

En el caso de una contribución como un texto. La aplicación puede dar la posibilidad de eliminar o modificar el texto (u otro tipo de objetos enviados), hay que tener en cuenta la posibilidad de incluir restricciones en este tipo de operaciones, por ejemplo, no sería lógico que cualquier usuario pueda modificar el contenido aportado por algún otro.

La mayoría de los chats sincrónicos no permiten corrección, se debe a una imposibilidad, dada por un problema de implementación. En los chats asincrónicos, en cambio, no existe esta restricción de implementación.

Pensemos cuan útil puede ser la posibilidad de editar en un chat sincrónico: si estamos ante un chat room en el que se esta dando una clase de, digamos, gramática, entonces puede ahorrase bastante tiempo y obtener mayor claridad, si el docente a cargo, pudiera editar los mensajes de los alumnos y corregirlos (tal vez resaltando las partes corregidas, o marcando los errores), de modo que no tenga que reenviar un mensaje explicando que " en tal mensaje anterior donde pusiste tal cosa debió ir tal otra"

R12. Registro de la Actividad (Logging Functionality)
El administrador de la aplicación puede requerir, poseer algún tipo de información referente a los eventos que se hayan producido durante alguna charla. Para ello la aplicación debería poder llevar un registro de los eventos en los cuáles se encuentre interesado el usuario.

R13. Proveer Información Estadística
Como funcionalidad adicional, la aplicación puede llegar a ofrecer al usuario información estadística acerca de su participación o la de algún otro usuario.

2.2 Requerimientos del Desarrollador

D1. Abstracción del mecanismo de comunicación

Un aspecto fundamental que debe cubrir el framework es evitar que el desarrollador deba preocuparse por la forma en que la información viajará de una máquina a otra, es decir la red comunicación subyacente entre los clientes de la aplicación debe ser transparente para el desarrollador.

D2. Administración de Sesiones

El usuario puede iniciar una nueva sesión cada vez que ingresa a la aplicación, o puede decidir ingresar a una sesión ya existente, la forma en que se procesa la información para manejar un mecanismo de múltiples sesiones deberá ser indiferente para el desarrollador.

D3. Reutilización de Componentes

El framework debe proveer la posibilidad de crear y extender las componentes de software, tanto del modelo como de la interfaz de la aplicación, pero a su vez, se pretende que pueda irse construyendo una biblioteca de componentes reutilizables. De forma de poder armar una aplicación sólo seleccionando los componentes adecuados.

D4. Registro dinámico de componentes

Es deseable, que los desarrolladores puedan trabajar desde diferentes lugares en la creación de nuevas componentes y que luego puedan registrarlas dinámicamente de modo de poder volverlas accesibles para cualquier otro desarrollador que desee incluirlas en su aplicación.

Particularmente la mayoría de los requerimientos del desarrollador no fueron tenidos en cuenta en la versión prototipo de Chatblocks. En el siguiente capítulo, analizaré dicha versión prototípica, y tras marcar los requerimientos que cubre, dejaré en suspenso aquellos otros aspectos que se irán cubriendo en cada uno de los capítulos posteriores al mencionado.

D5. Extensión del Sistema:

La comunicación puede enriquecerse con otros formatos audiovisuales, hoy día es común encontrar sistemas que soportan envió de voz y video. Se puede contar incluso con una pizarra compartida donde los participantes puedan "dibujar" sus ideas.

Aunque este tipo de aplicaciones escapa al alcance del dominio de esta tesis, es conveniente tenerlas en cuenta al momento de proveer un framework flexible que pueda ser incluido dentro de éste otro tipo de aplicaciones colaborativas.

D6. Deployment de Aplicaciones a través del Web

Una de las metas de esta implementación en Java del sistema, es poder brindar al usuario un mecanismo más flexible y accesible desde cualquier ubicación a través del ámbito Web. Esta es una de las principales razones para utilizar una tecnología tan difundida y aceptada en el desarrollo de aplicaciones para Internet como es Java

2.3 Síntesis de los Requerimientos del Sistema

Tabla 2.1. Síntesis de los Requerimientos del Sistema

	Requerimientos del Usuario
R1	Numero de Participantes
R2.	Tipo de Comunicación
R3.	Unidad de Envío de Información
R4.	Tipo de Almacenamiento
R5.	Perfiles de Usuario
R6.	Protocolo de Comunicación
R7.	Elección del Receptor
R8.	Semántica de las contribuciones
R9.	Organización de la Información
R10.	Awareness
R11.	Post-Edición
R12.	Registro de la Actividad
R13.	Proveer Información Estadística

	Requerimientos del Desarrollador
D1.	Abstracción del mecanismo de comunicación
D2.	Administración de Sesiones
D3.	Reutilización de Componentes
D4.	Registro dinámico de componentes
D5.	Extensión del Sistema
D6.	Deployment de Aplicaciones a través del Web

Capítulo 3. Prototipo de Chatblocks empleando COAST

El prototipo inicial de Chatblocks fue implementado utilizando un framework anterior al desarrollo de DyCE (el entorno de colaboración dinámico en el que esta basado la implementación final de Chatblocks) llamado COAST (Co-Operative Application System Technology), el cual esta íntegramente implementado en Smalltalk.
Así mismo, gran parte del diseño general de DyCE esta sustentado en las ideas principales que utilizaron los desarrolladores de COAST.

En la siguiente sección pretendo dar una breve reseña de cómo es la filosofía de COAST, motivado por el hecho de que así se podrá explicar con mayor claridad en que consiste y como se llega al diseño actual de Chatblocks implementado para DyCE.
El uso de COAST o DyCE, tiene incidencia directa en los requerimientos del desarrollador planteados en el capitulo 2.

3.1 COAST

El prototipo inicial se hizo siguiendo el paradigma de programación proporcionado por COAST.
El framework COAST pretende hacer el desarrollo de groupware sincrónico más fácil, con un costo comparable al desarrollo de aplicaciones monousuario equivalentes.

COAST proporciona una arquitectura de groupware básica, un ambiente para la construcción de aplicaciones groupware, y una metodología subyacente para el desarrollo de groupware sincrónico.

Básicamente al utilizar COAST se obtienen los siguientes beneficios

a. La replicación de objetos es transparente.
b. Consistencia de los datos (a través de un mecanismo de transacciones).
c. Actualización automática de views cuando un objeto compartido cambia.

Para conseguir esto COAST debe:

a. Ocultar al programador de la aplicación, la red, es decir, el programador de la aplicación no debe realizar acciones de transporte de objetos sobre la red. El sistema decide que objetos tienen que ser transferidos por la red y cuando debe suceder esto.

b. Ocultar al programador de la aplicación, la ubicación física de los objetos. Los objetos compartidos tienen un ID global único. Donde se localizan físicamente objetos compartidos, es transparente al programador de la aplicación. El sistema decide, por cada objeto compartido, en que situaciones debe mantener una o más réplicas sincronizadas.

c. Ocultar parcialmente la concurrencia. COAST incluye un sistema de transacciones. Mediante el uso de transacciones, el programador de la aplicación, puede manejar cómodamente la concurrencia global. El sistema asegura que todas las transacciones sean serializables.

3.1.1 Extensión del Paradigma Model View Controller (MVC)

La GUI de una aplicación constituye lo que conocemos como Views y Controllers.

Los objetos GUI son responsables de presentar un documento en forma gráfica e interpretar el ingreso de datos del usuario.

Los objetos documento son responsables de representar la información esencial del modelo. En otras palabras, ellos representan la información que se guardaría en aplicaciones monousuario, esta generalmente es un archivo.

Los objetos GUI son objetos locales, mientras que los objetos modelo se consideran objetos compartidos.

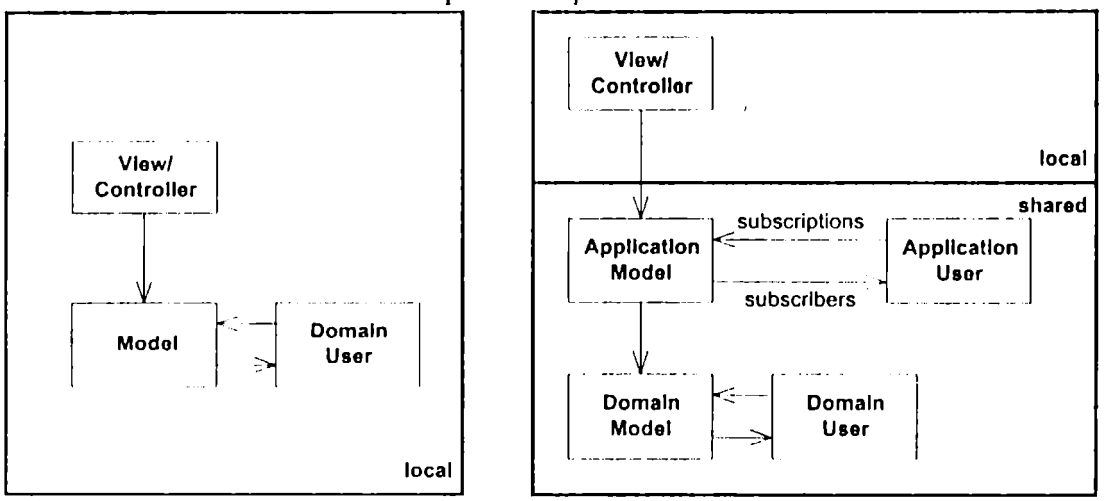
COAST extiende esta notación para trabajar con objetos modelo compartidos:
Básicamente una aplicación COAST deberá constar de un modelo compartido (Shared Application Model), y de una vista de la lógica del modelo compartido (Local Application Model), el cual conforma el GUI

3.1.2 Modelado de aplicaciones groupware

Para mantener una consistente y flexible visualización de los objetos compartidos, se han extendido conceptos como el MVC para el uso en aplicaciones groupware. Gracias a esto, es posible definir un modelo de aplicación uniforme para aplicaciones multiusuario en un

nivel más abstracto. Este modelo tiene que considerar aspectos específicos del groupware, como *awareness* o *floor control*.

COAST transfiere este concepto a las aplicaciones colaborativas.



Modelo convencional (MVC) *Modelo COAST*

Figura 3.1. Modelos de arquitectura

Comparación entre el modelo convencional, para aplicaciones monousuario, y el modelo COAST, para aplicaciones groupware. Hay diferencias significativas entre como trabajar con los objetos compartidos, al accederlos desde la interfaz local de usuario, es necesario un mecanismo transaccional que asegure la consistencia de la información.

3.1.3 Componentes de la arquitectura

COAST incluye dos tipos de programas que juegan papeles específicos en la arquitectura del sistema distribuido del framework.

Manipulación de clientes COAST

COAST emplea una arquitectura distribuida y replicada: Cada usuario interactúa con una instancia individual de la aplicación desarrollada en base a COAST, el *COAST Client*. Los clientes COAST normalmente proporcionan una interfaz al usuario final que permite manipular datos de la aplicación activamente.

COAST Mediator – sincronización y almacenamiento

Los mediadores COAST negocian la comparación sincrónica de los datos de la aplicación. Éstos mantienen la copia primaria de los datos que son compartido entre sus clientes. Los datos de la aplicación necesarios para ejecutar una aplicación COAST son cargados y replicados transparentemente bajo demanda de un COAST mediator en un cluster. El papel del servidor COAST es similar al de un servidor de WEB/FTP, recuperar datos de un almacenamiento persistente y enviarlo por la red al cliente solicitante. El COAST mediator

provee servicios adicionales creando nuevos clusters o volúmenes en su namespace. El mediador COAST es un componente genérico y como a tal independiente de los datos específicos de la aplicación.

3.1.4 Requerimientos del Desarrollador

COAST emplea un mecanismo automático de sesiones que pueden ser recuperadas desde el servidor para continuar trabajando con los datos compartidos que uno haya estado modificando con anterioridad. Por lo tanto al emplear COAST, queda resuelto el requerimiento D2 del desarrollador. Por otro lado, COAST se ve limitado en cuanto a la carga dinámica de componentes (D4), debido a que para que una componente pueda ser utilizada es necesaria copiar la imagen (se utiliza una imagen Smalltalk como aplicación) en cada uno de los clientes que deseen correr la aplicación. La reutilización no es a nivel de componentes (requer. del desarrollador D3) ya que COAST no ofrece en un sentido explícito componentes para distribuir sino que la reutilización es al nivel estándar del paradigma de objetos, es decir, con la Clase como el bloque más extenso y cerrado posible.

3.2 Arquitectura General del Prototipo

Aunque el diseño actual de Chatblocks mantiene la esencia del prototipo inicial, la implementación pasó a ser totalmente diferente, no solo por el cambio de COAST a DyCE, sino fundamentalmente por el cambio en el paradigma de programación utilizado: mientras COAST es 100% implementado en Smalltalk. DyCE por su parte esta totalmente implementado con tecnología Java, pensando mas en las capacidades de distribución que puede brindar Internet, el framework DyCE incluye un servidor http con lo cuál vuelve aún más accesibles a las aplicaciones desarrolladas con el mismo a través de un web site, por ejemplo.

La arquitectura original, por ser una aplicación COAST, consta de una parte compartida o Shared y otra exclusiva por cada usuario o Local, lo cual significa que en cada máquina que está ejecutando la aplicación existen dos clases de objetos los locales y los compartidos. Los mismos se describen a continuación:

Tabla 3.1. Ambientes de ejecución de los componentes Chatblocks

Componentes COAST	Componentes Chatblocks	Ambiente de Ejecución
Domain Model	Messages Pool	Shared
Shared Application Model	Managers	Shared
Local Application Model	Views	Local

Como lo indica la tabla, en la parte compartida se encuentra el MessagesPool y los managers, y en la parte local las views.

El Domain Model queda representado por el siguiente diagrama clases:

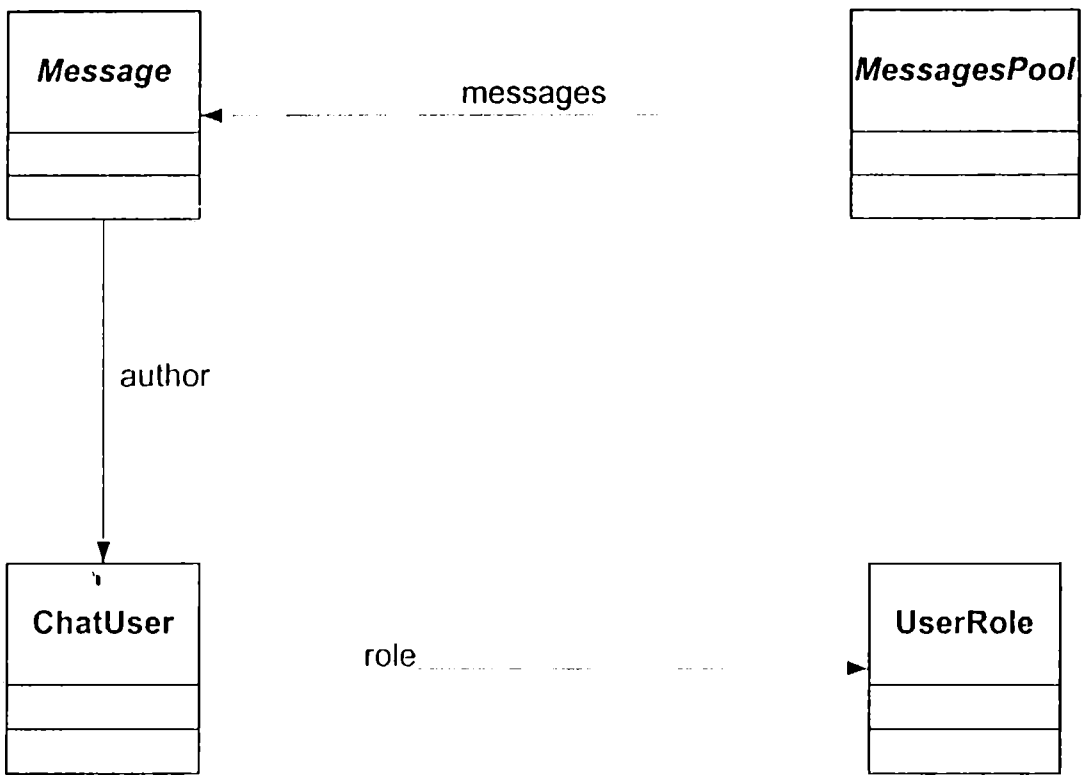


Figura 3.2. Domain Model prototípico

Chatblocks aporta la estructura básica sobre la que se define los componentes específicos para el tipo de aplicación que se quiere construir. En esta estructura, los componentes intercambiables son los managers y sus respectivos Views. Es decir, de acuerdo al tipo de comunicación o discusión que se quiera implementar, se deberá optar por uno u otro manager.

Chatblocks cubre los requerimientos mencionados de dos formas:

Hotspots: Son aquellos aspectos personalizables en cada aplicación final. El programador de las aplicaciones chats creadas con Chatblocks podrá elegir una de las variantes proporcionadas por el framework para cada uno de estos aspectos.

Frozenspots: Estos aspectos serán fijos para todas las aplicaciones finales. El framework posee estos aspectos fijos como resultado de decisiones de diseño del mismo, ya que se deben fijar algunos aspectos para no perder eficiencia.

El prototipo tuvo en cuenta los siguientes aspectos (cada uno de ellos será tratado nuevamente cuando se presente la versión final de Chatblocks en el Capítulo 6):

Unidad de Envío

Chatblocks considera como una unidad el envió por mensajes (es decir por grupo de caracteres, en lugar de considerar a cada carácter como la unidad de información transferible).
Se opto por fijar este aspecto ya que el framework esta pensado para realizar aplicaciones chat que provean floor control y algún tipo de semántica. Para ello se necesita controlar el flujo de mensajes. Con caracteres, como unidad de envío, es muy difícil determinar cuando termina un mensaje y cuando empieza el otro. Ya que un *punto* o un *enter* podría ser interpretado como una separación de párrafos o finalización de mensajes, agregando confusión y perdida de expresividad al usuario. Con esto se fija un frozenspot para el requerimiento de usuario R3

Nro. de Participantes

Chatblocks esta pensado para la creación de chats grupales donde existen mas de dos participantes, sin embargo no impone restricciones en este aspecto, permitiendo que dicha característica pueda ser determinada por el administrador del chat, si es que desea tener un número máximo o fijo de participantes. Esto abarca el requerimiento de usuario R1

Almacenamiento
Permanente

Chatblocks se encargará de almacenar permanentemente los mensajes emitidos por los participantes del chat. Los mensajes residirán en un servidor o recurso compartido, permitiendo que todos los usuarios los accedan en el momento que lo deseen. Para esto, el prototipo de Chatblocks se basa en el manejo de sesiones provisto por COAST, en donde el usuario antes de ingresar al sistema, puede decidir si lo hará en una sesión ya existente o si decidirá empezar una nueva sesión. Este tipo de situaciones también es contemplado en DyCE, pero con ciertas diferencias en cuanto a la capacidad del usuario de elegir la sesión con la que desea comenzar a utilizar la aplicación. El tipo de almacenamiento volátil no es tenido en cuenta, pero se lo puede tratar como un caso especial, en donde el usuario siempre entra a la aplicación en una sesión nueva. Esto conforma el Requerimiento de Usuario R4.

Receptor
Las situaciones a las que apunta emular Chatblocks son conversaciones en las cuales varias personas se juntan da discutir un tema dado, por ese motivo, cuando una persona realiza un aporte al chat, este debería ser reflejado en todos los clientes de la aplicación. Por esa razón los mensajes poseen a todos los participantes del chat como receptor. Esta postura se mantiene igual en la versión final. Sin embargo el desarrollador es totalmente libre de construir una aplicación en la que exista la posibilidad de crear canales privados entre dos usuarios. Este punto viene a contemplar el requerimiento R7.

Desarrollo de la conversación
Poder contar con almacenamiento permanente de mensajes y manejo de sesiones, deviene en la posibilidad de albergar los tipos de comunicación existentes: sincrónica y asincrónica.

Clasificación de Usuarios
El framework permitirá soportar usuario con diferentes roles o que ninguno cumpla un rol especial dentro de la aplicación. El programador podrá elegir una de estas dos modalidades.

El hecho de que existan roles está estrechamente relacionado con el Floor Control, ya que implica que cada usuario podrá realizar diferentes acciones.
Por ejemplo en el caso de una aplicación de Ajedrez, existirá un usuario con el rol: Negras y otro con el rol: Blancas. La aplicación sabrá cual comienza la jugada por el rol que asume cada participante. Análogamente en una simulación de un debate, bastará con definir el rol de cada uno de los participantes, para que la aplicación pueda arbitrar el debate cediendo el turno en que cada uno puede realizar aportes a la discusión.

Esta característica definición de roles usuario, permite establecer los perfiles de usuario, contemplados por el requerimiento R5.

Floor Control
Este es el aspecto más importante que desea atacar el framework. El programador del chat, podrá optar entre uno de los protocolos provistos por el framework, o la ausencia de este. Si se desea recrear una conversación real, se debe tener en cuenta la gran cantidad de situaciones que existen en las mismas, es decir, no es lo mismo participar de una clase, donde existe un profesor que la dirige y exige a los alumnos a realizar las preguntas en forma ordenada u de a uno a la vez, que participar de una conversación informal donde todos hablan a la vez.

Para proveer flexibilidad en realizar aplicaciones chat que emulen ciertas situaciones reales se debe proveer algún mecanismo de floor control. En el caso de un sistema chat, el floor control es lo que determina el protocolo de la comunicación o tipo de charla que se pretende realizar con la aplicación (Esto viene a cumplir con el requerimiento R6)

La mayoría de las aplicaciones cooperativas (groupware) poseen Floor Control, pero los chats comerciales no. Generalmente el protocolo de comunicación es implícito y queda a decisión de los participantes de la charla el cumplirlo

Semántica de los Mensajes
Ninguna
Respuesta a un mensaje
Referencia a un Objeto

Con el framework se podrán crear chat cuyos mensajes posean semántica o no. Los mensajes que poseen semántica podrán ser una respuesta a un mensaje anterior, o podrán referenciar algún objeto.
Este es otro aspecto que ha de ser explorado, ya que la mayoría de los chats comerciales no hacen hincapié en este aspecto. (Requerimiento de Usuario R8)

Visualización de los mensajes
El framework proveerá dos modalidades: una lista de mensajes, o para el caso en que los mensajes aporten cierta semántica extra como es el caso de una threaded chat, se provee la visualización con estructura de árbol, cabe destacar que este aspecto esta directamente relacionado al ordenamiento de los mensajes y a la semántica de los mismos, ya que esos criterios serán tomados en cuenta al momento de la visualización de la estructura seleccionada.
El programador podrá optar por fijar una de estas opciones, o permitir el intercambio de las mismas en tiempo de ejecución, dependiendo de los requerimientos de la aplicación a construir (R9: Organización de la Información)

En este punto, la limitación de poseer solo dos tipos de vista puede ser solucionado proveyendo un diseño extensible que permita la creación de cualquier tipo de vista de los mensajes.

Awareness
En la versión COAST, se pensó en proveer dos tipos de awareness:
de participación y de conexión.
El awareness es uno de los aspectos más importantes de las aplicaciones cooperativas.
Las aplicaciones chat creadas con el framework constarán de un grupo de participantes, los mismos podrán conectarse en diferentes momentos.
Chatblocks proveerá awareness de participación, el mismo muestra quieren son los usuarios que participan del chat, en cambio el awareness de conexión, mostrará cuales son los participantes que se hallan conectados a la aplicación en un momento determinado.

Este es otro de los aspectos que se pretenden mejorar en la versión Java, incluyendo además de awareness de conexión y participación, un awareness de actividad, de modo de poder tener una noción mas profunda de que es lo que se encuentra haciendo en un determinado momento un usuario: como por ejemplo, saber si solo esta leyendo los mensajes recibidos, o si esta componiendo uno nuevo. Esto complementa el requerimiento de usuario R10

Post-Edición

Chatblocks en su versión prototipo provee la eliminación de mensajes previamente enviados. Es decir permitirá deshacer acciones realizadas por los usuarios. Los mensajes sólo podrán ser borrados por sus autores, esto asegura que las acciones realizadas por un usuario no sean anuladas por otro.

Sin embargo, también puede ser muy útil el poder editar un mensaje ya emitido. De esta forma, si estamos ante un sistema chat que permita llevar adelante una clase virtual de, por ejemplo, aprendizaje de lenguajes, el usuario con el rol de profesor, puede tener la posibilidad de corregir y resaltar los errores hechos por algún alumno, de manera sencilla y natural, sin necesidad de que copiar, editar y reenviar todo como un nuevo mensaje. Este es un aspecto contemplado en la versión final en DyCE y apunta a resolver el requerimiento de usuario R11.

Extensiones

Las aplicaciones creadas con Chatblocks están diseñadas para ser sub-aplicaciones de una herramienta de comunicación. Por ejemplo un chat creado con Chatblocks puede ser parte de una herramienta de interacción que tenga un chat y una pizarra de dibujo cooperativa.

Sin embargo, en este aspecto la aplicación se ve limitada en su portabilidad y escalabilidad, ya que siguiendo con la filosofía COAST, esta considera a la aplicación como un todo, y se vuelve difícil intentar utilizarla como una componente cerrada de software. Esto compromete en parte el requerimiento del desarrollador D5.
Es en este punto donde la utilización de DyCE deja en evidencia sus virtudes mas poderosas, al incluir el concepto de "Componentes Groupware", noción que es explicada en el capítulo siguiente.

3.2.1 Interacción entre los componentes del Framework

Cada manager cubre uno o más aspectos de la aplicación final, los managers interactúan entre sí, para llevar a cabo las operaciones específicas dentro de la aplicación final. Estos realizan operaciones directamente sobre el MessagesPool, que compone el almacén de mensajes, agregando o extrayendo de este, los mensajes que necesiten para completar las operaciones.

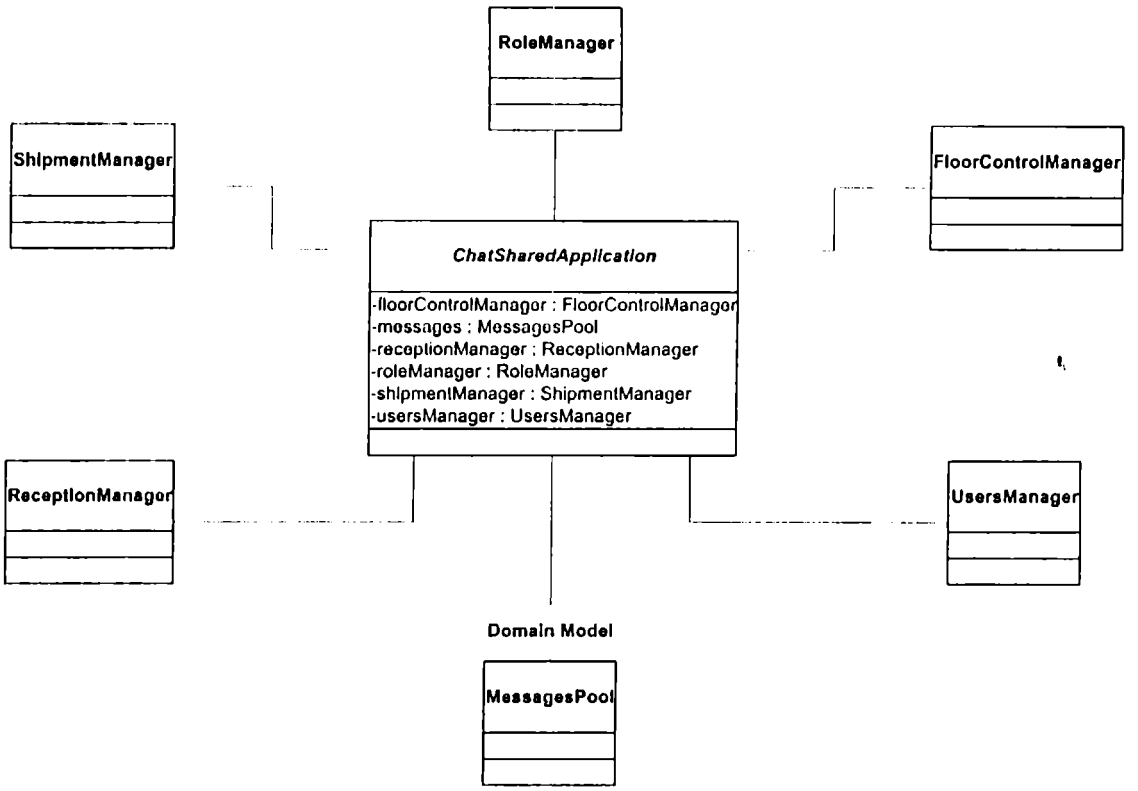


Diagrama 3.1 Prototipo COAST: Shared Application Model

Las views poseen una comunicación directa con los managers, reflejando los cambios que sufre el corazón de la aplicación, por ejemplo, nuevos mensajes, ingreso y egreso de participantes, cambios de turnos, etc. Cada view interactúa con un manager específico, de éste recibe la información que debe mostrar al usuario y hacia éste envía las operaciones que el usuario ordena a la aplicación.

De este modo, los Views pueden esquematizarse en el siguiente diagrama de clases:

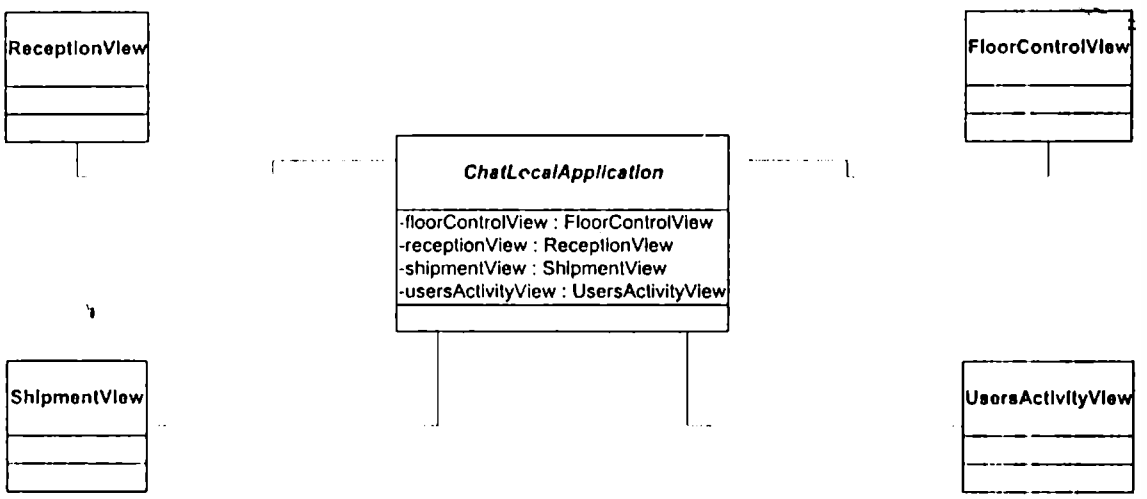


Diagrama 3.1 Prototipo COAST: Local Application Model

La interfaz del usuario surge de la composición de views, mientras que el corazón de la aplicación, la parte compartida, surge de la composición de managers y el MessagesPool.

En el siguiente esquema puede verse la interacción entre la parte local y la compartida del framework. En el caso del RoleManager, no es necesaria una representación visual del mismo.

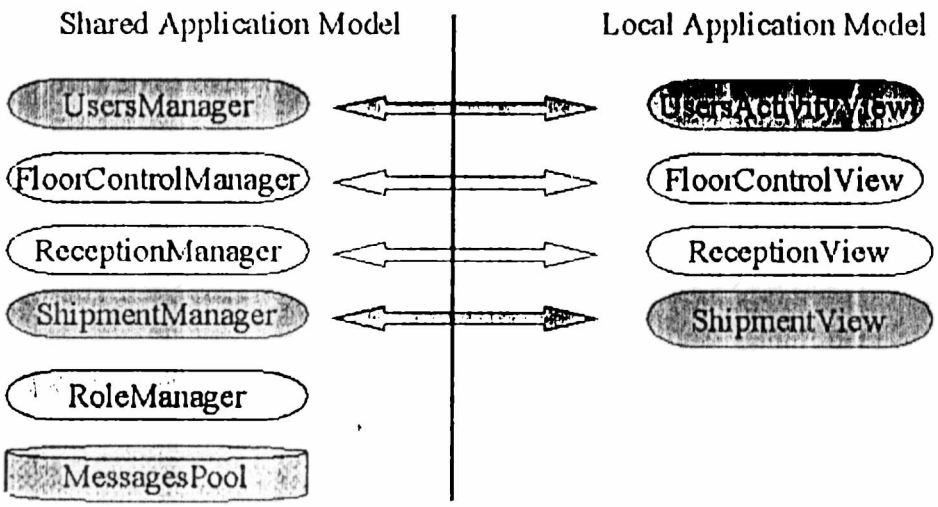


Figura 3.1. Componentes del Framework

Puedo resumir la funcionalidad a cargo de cada uno de los managers, explicando cuales de los requerimientòs de usuario satisface cada uno:

UsersManager
Se encarga del número de participantes y awareness de actividad de los mismos (R1 y R10). En caso de poder definirse perfiles de usuario especiales, como por ejemplo un usuario administrador, recae en este objeto la responsabilidad. En realidad en el prototipo no ha sido contemplado el tener roles y perfiles de usuario, tal vez sin necesidad de ser una relación biunívoca entre ambos, es decir, todos los usuarios podrían tener el mismo rol, pero poseer diferentes perfiles de usuario, por ejemplo: el poseer un usuario administrador, puede ser independiente de que el protocolo de la discusión contemple un rol con esta función, es mas esto es muy común en los chats tradicionales, donde no hay roles, pero si existe un administrador.

FloorControlManager
Establece el protocolo de la comunicación del sistema (involucra R2, R6 y R7)

ReceptionManager
Se encarga de administrar la información enviada, decide que mensajes serán accesibles y cuales serán filtrados. A diferencia del resto, hay que tener en cuenta que el ReceptionManager se encarga de la organización de la información en el ámbito general (R9), y su contraparte local, el ReceptionView, es la que aplica filtros y configuraciones locales para cada usuario (R9 y de cierta forma se ve involucrado en R11)

ShipmentManager
Es el encargado de completar, chequear la correctitud y determinar la viabilidad para enviar un mensaje. Obviamente se ve involucrado con la semántica de las contribuciones y con el protocolo de la comunicación (R8 y R6)

RoleManager
Se encarga de manejar los roles de usuario, es decir de la asignación de roles de acuerdo a al protocolo que se haya definido (R7).
El MessagesPool es el encargado de mantener todos los mensajes enviados, se lo puede ver como un almacén de mensajes de carácter permanente o no de acuerdo al tipo de manejo de sesiones que se emplee (R4)

El propósito principal del framework es facilitar al programador de una serie de alternativas para combinar los componentes existentes o en todo caso idear los propios, basándose en los componentes de base provistos. Pero tal libertad para combinar componentes y obtener así una aplicación final, esta limitada por el hecho de que existen relaciones que deben ser respetadas. Si bien, el framework trata de dejar lo mas independientemente posible este tipo de relaciones, existen, y se hacen mas o menos fuertes de acuerdo al tipo de sistema que se quiere construir.

Debido a que la extensión del framework es a través de la sub-clasificación de las componentes básicas, esto permite que todo el mecanismo de comunicación (como hacer para que un mensaje llegue al MessagesPool, y luego pueda ser accedido por todos los

clientes) pueda ser abstraído en dichas componentes básicas. Con esto nos aseguramos uno de los requerimientos del desarrollador: abstracción del mecanismo de comunicación (D1)

La estructura presentada aquí constituye la base de la construcción final de Chatblocks en DyCE.

3.2.2 Síntesis de la Arquitectura General del Framework

El framework provee distintas variantes para cada tipo de componente, formando una familia de componentes del mismo tipo.
Para realizar una aplicación que emule una conversación determinada basta solo con incluir los componentes que satisfagan las situaciones requeridas por la aplicación. Por ejemplo, si necesitamos realizar un chat que emule un debate de TV, deberíamos incluir un componente que otorgue turnos alternados, pero si en cambio queremos emular una discusión de una conferencia de prensa, deberíamos incluir un componente que seleccione entre los participantes con el pedido de palabra y se la otorgue a uno de ellos.

El framework se divide en dos capas una local llamada (Local Application Model) que se compone de todas las partes locales de la aplicación, tales como interfaces y eventos. Y una parte compartida (Shared Application Model) en donde se encuentran las componentes a compartir por todos los usuarios.

En la capa compartida se encuentran los managers, cada manager cubre uno o más aspectos de la aplicación final, los managers interactúan entre sí, para llevar a cabo las operaciones que se les encomiendan. Estos realizan operaciones directamente sobre el MessagesPool, que es el repositorio de mensajes, agregando o extrayendo de este, los mensajes que necesiten para completar las operaciones.

Las views poseen una comunicación directa con los managers, reflejando los cambios que sufre el corazón de la aplicación, por ejemplo, nuevos mensajes, ingreso y egreso de participantes, cambios de turnos, etc. Cada view interactúa con un manager específico, de éste recibe la información que debe mostrar al usuario y hacia éste envía las operaciones que el usuario ordena a la aplicación.
La interfaz del usuario surge de la composición de views, mientras que el corazón de la aplicación, la parte compartida, surge de la composición de managers y el MessagesPool.

3.2.3 Síntesis de Requerimientos

Se puede resumir los requerimientos que se cubren con esta estructura en la siguiente tabla:

Tabla 3.2. Requerimientos del Usuario

	Requerimientos del Usuario	Funcionalidad delegada en
R1	Numero de Participantes	UsersManager
R2.	Tipo de Comunicación	FloorControlManager
R3.	Unidad de Envío de Información	Decisión de diseño: mensaje
R4.	Tipo de Almacenamiento	Almacenamiento permanente provisto por COAST
R5.	Perfiles de Usuario	UsersManager
R6.	Protocolo de Comunicación	FloorControlManager, RoleManager
R7.	Elección del Receptor	FloorControlManager
R8.	Semántica de las contribuciones	ShipmentManager, FloorControlManager
R9.	Organización de la Información	ReceptionManager, ReceptionView
R10.	Awareness	UsersActivityView, FloorControlView, limitados a awareness de presencia y conexión
R11.	Post-Edición	No contemplada en el prototipo.
R12.	Registro de la Actividad	No contemplada en el prototipo.
R13.	Proveer Información Estadística	No contemplada en el prototipo.

Tabla 3.3. Requerimientos del Usuario

	Requerimientos del Desarrollador	Factibilidad utilizando COAST
D1.	Abstracción del mecanismo de comunicación	Provisto por la estructura abstracta del prototipo
D2.	Administración de Sesiones	Automáticamente por COAST
D3.	Reutilización de Componentes	Solo al nivel de clases, POO tradicional
D4.	Registro dinámico de componentes	No disponible con COAST
D5.	Extensión del Sistema	A nivel del paradigma orientado a objetos
D6.	Deployment de Aplicaciones a través del Web	No soportado directamente por COAST

Los puntos inconclusos pretenden ser cubiertos con la versión DyCE del framework.

Principalmente, en el siguiente capítulo, donde se presente que son Componentes Groupware, haré hincapié en las falencias presentes en un sistema como COAST, sobre todo en los puntos D3, D4, y D5 de los requerimientos del desarrollador.

Capítulo 4. Componentes Groupware

Uno de los conceptos centrales provistos por DyCE es el de "Componentes Groupware", en este capítulo intentaré explicar en que consiste y como ayuda esto al desarrollo de un framework mas flexible y reutilizable.

4.1 Aplicaciones Cooperativas

Las aplicaciones CSCW (Computer-Supported Co-operative Work) permiten a los grupos de usuarios (normalmente distribuidos sobre el tiempo o el espacio) a resolver cooperativamente una tarea en común por medio de la cooperación de un conjunto de objetos compartidos, tales como documentos, proyectos gráficos, herramientas para construcción de modelos, etc.
El área de CSCW involucra el uso de computadoras en la configuración del grupo. Las herramientas basadas en computadoras utilizadas para el soporte del trabajo colaborativo son también denominadas groupware.
La definición de Coleman del uso de groupware dice que "groupware permite que equipos de persona y otros paradigmas que requieran de gente puedan trabajar juntos, tanto en tiempo como en espacio" ([Col97], p.1).
Greenberg [Gre91] define el término groupware como:
Groupware es el software que soporta y aumenta el trabajo en grupo. Es una palabra técnicamente orientada para diferenciar productos "orientados a grupos", explícitamente diseñados para asistir a grupos de gente a trabajar en conjunto de productos mono-usuarios que ayudan a las personas solo en tareas aisladas desvinculadas con el resto del grupo.
Para mi punto de vista, tendríamos que pensar en groupware como cualquier medio que permite mejorar el trabajo en grupo (incluso el correo postal tradicional puede entrar en esta categoría). Mientras que para aquellas herramientas groupware que consisten en aplicaciones de software, podemos considerarlas como CSCW.

El objetivo de los sistemas CSCW es ayudar no sólo a cubrir la acción cooperativa de las tareas sino también la comunicación entre los integrantes del grupo así como también la

coordinación de las tareas realizadas individualmente en un grupo. Esto lo hace proveyendo a los usuarios de aplicaciones colaborativas. [DCS94] define ampliamente una aplicación colaborativa de esta forma: "Una aplicación colaborativa es una aplicación de software que a) interactúa con múltiples usuarios, esto es, recibe entradas de múltiples usuarios y genera salidas a múltiples usuarios, y b) acopla las interacciones entre estos usuarios, es decir, permite que la entrada de un usuario influencie la salida mostrada en otro usuario". Esta definición general también incluye al sistema colaborativo que se presenta en esta tesis.

Es ampliamente aceptado que para el éxito de las aplicaciones groupware se necesita proveer soporte para lo que fue denominado "las tres C de workgroup computing"[The01]:

- comunicación: permite el intercambio de por ejemplo ideas, notas durante el transcurso del trabajo;
- coordinación: permite el flujo estructurado de los elementos involucrados entre cada tarea, así como también significa armar la agenda y controlar las actividades cooperativas complejas;
- cooperación: provee herramientas para trabajar en conjunto en un objeto en común.

4.2 Arquitecturas basadas en componentes

Incluso en los sistemas gráficos multi-windows de hoy día, muchas aplicaciones colaborativas son desarrolladas y presentadas a los usuarios como un único bloque con un gran número de diversas funcionalidades. Tales aplicaciones pueden ser difíciles de mantener y poco flexibles, debido al hecho de que contienen muchas interrelaciones e interdependencias hard-coded (cableadas en el código del programa). Además, las tecnologías empleadas para desarrollarlas no son suficientes para cubrir los requerimientos de flexibilidad y extensibilidad.
Arquitecturas de componentes tales como OLE32, DCOM, OpenDOC o más recientemente JavaBeans ayudan a componer aplicaciones de componentes reutilizables e inter operables. Las diferentes aproximaciones a la creación de componentes difieren en el nivel que las mismas exponen a los usuarios el hecho de que los sistemas que están utilizando están construidos en un tipo de arquitectura orientada en componentes.

Esta tesis involucra un framework basado en DyCE, por lo tanto las aplicaciones desarrolladas con el mismo terminarán siendo componentes groupware.
Para entender las principales diferencias entre las distintas propuestas a este tipo de componentes de software, es importante discutir la distinción entre aplicaciones, suite de aplicaciones, framework de componentes y el sistema de Componentes Groupware empleados por DyCE y en el que se pretende desarrollar Chatblocks. Las principales dimensiones de distinción son los diferentes grados de facilidad, integración y extensibilidad provistos por la propuesta.

Las **aplicaciones** son percibidas como ambientes cerrados que proveen una cierta funcionalidad al usuario. Para extender la funcionalidad, una aplicación puede proveer la

posibilidad de invocar (ejecutar) otra aplicación, tal como un editor que es utilizado para escribir texto, puede permitir la invocación directa de por ejemplo un traductor de lenguajes. La unidad de interacción, sin embargo, recae en una sola aplicación. La llamada a otra aplicación (y opcionalmente el pasaje de cierta información, como por ejemplo, el texto a traducir) no constituye una integración firme. Mas aún, esta forma de integración no es realmente extensible. Adicionalmente, esta forma de integración no será mostrada al usuario como una única aplicación que provee un amplio rango de funciones. En su lugar, el resultado es siempre verse como un conjunto de aplicaciones diferentes (cada una con su propio look-and-feel, etc).

Las **suites de aplicaciones** consisten de un número de aplicaciones, que son diseñadas para ser integrables en una forma que dan la sensación al usuario de trabajar con una aplicación sola. Un ejemplo de una suite de este tipo es el Netscape Communicator, la cual esta compuesta de un navegador web, un editor de HTML, una herramienta de mensajería instantánea e incluso un calendario. La suite Communicator es desarrollada en forma tal que alternar entre las distintas aplicaciones es hecho de forma estándar, vía un menú, por ejemplo.
Típicamente, las suites de aplicaciones también refuerzan la idea de tener un único look-and-feel sobre todas las herramientas de la suite, y también define una forma estándar para intercambiar la información entre las mismas. Para el usuario, la suite aparece como una sola gran aplicación. La mayoría de las suites de aplicaciones, son sin embargo, no extensibles. La misma puede ser restrictiva, por ejemplo, el usuario puede decidir que herramientas quiere instalar, pero generalmente no es posible la integración firme en la estructura de la suite de otras aplicaciones.

Los **frameworks de componentes** proveen una base extensible dinámicamente para la integración de diferentes herramientas (incluso de diferentes proveedores), de acuerdo a las demandas y requerimientos del usuario. Uno de los frameworks de componentes más usados es sin dudas el OLE2 (véase [Cha96]). Este framework de componentes es utilizado para integrar documentos de una aplicación en documentos de otra aplicación (por ejemplo: presentar fórmulas matemáticas avanzadas dentro de un documento producido con un procesador de texto), comúnmente, existe un OLE2 container (en el ejemplo, el procesador de texto), el cuál incorpora una componente OLE2 (en este caso el editor de fórmulas matemáticas).
El acceso a las componentes disponibles y su posterior integración es vía mecanismo estándares, controlados por el usuario (vea la figura 4.1).
Teniendo en cuenta que se desarrolle de acuerdo a las especificaciones del framework de componentes OLE2, cualquier aplicación puede servir como un OLE2 container o como una componente OLE2. Los componentes pueden incluso anidarse jerárquicamente. El usuario puede extender el sistema, simplemente instalando nuevas aplicaciones que sirvan como OLE2 containeres o como componentes. Estas nuevas aplicaciones pueden ser también usadas dentro de otras aplicaciones OLE2 (las cuales no necesitan tener conocimiento adicional acerca de tales nuevas aplicaciones). La combinación de aplicaciones OLE2 en un documento a menudo se presenta al usuario como una sola y poderosa aplicación. Los contenidos del documento son directamente ligados a la aplicación específica necesaria para editar dicho contenido. Si la aplicación requerida no está presente en el sistema, el contenido incluido no puede ser editado. El mapeo entre los

datos incluidos en el documento y la aplicación requerida para editarlo no puede ser modificado por el usuario. Si, por ejemplo, el usuario A recibe un documento conteniendo una fórmula matemática creada usando una herramienta XYZ, pero prefiere usar otra aplicación más intuitiva, digamos ABC, no podrá hacerlo. Este tipo de situaciones, presente con estos tipos de frameworks de componentes se vuelven aún más evidentes en configuraciones colaborativas, donde la opción de un usuario influencia el trabajo de los otros. Debido a que no existe un lugar central desde el cual los componentes pueden ser recuperados e instanciados, un usuario podría introducir un componente tal que los otros usuarios no tengan disponibles, con lo cual puede provocar una gran ruptura en la colaboración existente entre los miembros del grupo.

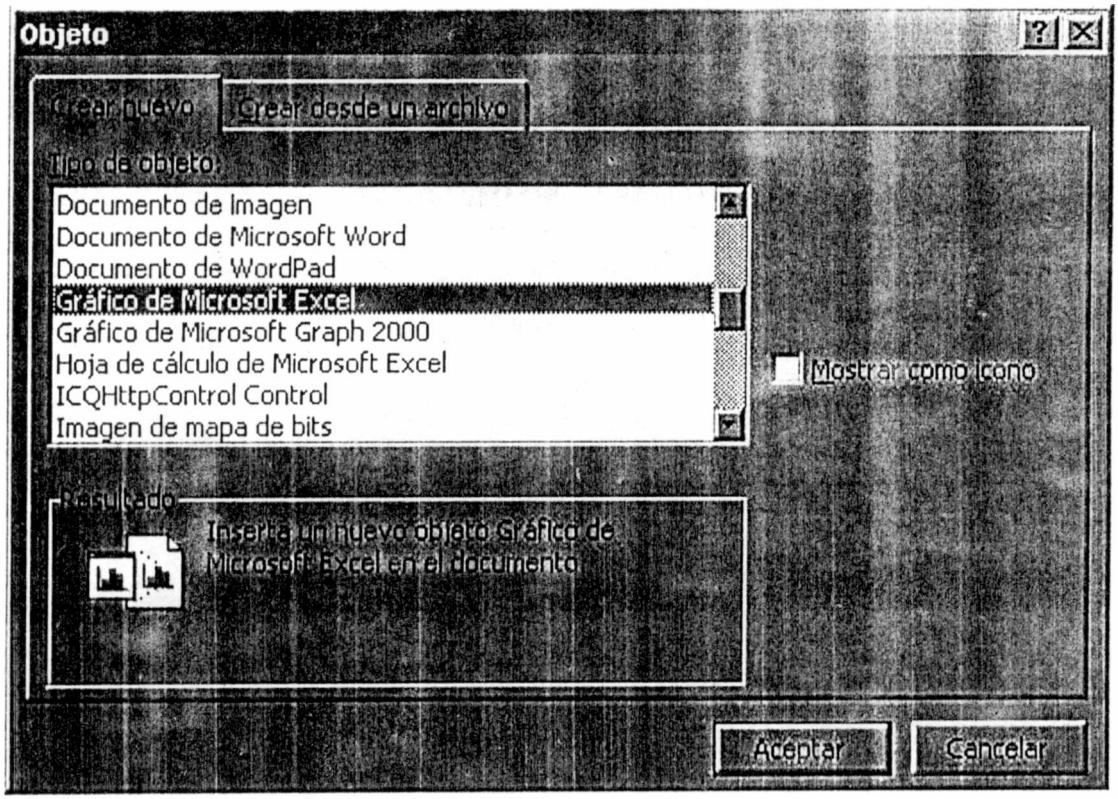


Figura 4.1. Integración de un objeto OLE en un procesador de texto

4.2 Dificultades que presenta un sistema como COAST

COAST ([SKSH96], [SSS99])separa la implementación del comportamiento de la aplicación y el modelo de datos compartidos. La aplicación colaborativa necesita estar presente en todos los lugares, donde se pretenda ejecutar un cliente. Los datos son replicados entre los sistemas. Un servidor central se encarga de la persistencia, el acceso a objetos compartidos y del soporte para colaboración asincrónica (el COAST Mediator).

Las aplicaciones realizadas con COAST no son basadas en componentes, al menos no en el sentido que emplea DyCE. COAST es un framework de desarrollo y como tal es extendido en aplicaciones específicas por los programadores utilizando los hot-spots provistos por el

framework. Las aplicaciones resultantes no son reutilizables o combinables. COAST no provee soporte para la extensión del sistema colaborativo en tiempo de ejecución, o personalización por parte del usuario final. Esto limita en gran medida la capacidad de desarrollar con Chatblocks un sistema que pueda ser fácilmente integrado en sistemas mayores, o que pueda ser reconfigurado de acuerdo a las decisiones del usuario. Al emplear DyCE, toda aplicación final creada con Chatblocks termina siendo una componente DyCE.

Las aplicaciones COAST no permiten al usuario extender las sesiones colaborativas con herramientas adicionales, el sistema no provee soporte para seleccionar que herramientas emplear, en que situaciones o para que tipo de documentos.

COAST puede ser visto como un ambiente de desarrollo groupware no extensible, es decir no hace previsiones para el deployment de nuevas herramientas colaborativas.

Además como la mayoría de los sistemas groupware existentes, no soporta el uso de componentes del lado del servidor, las cuales puedan ser invocadas en conjunción con las componentes colaborativas del lado del cliente para proveer una funcionalidad particular.

4.3 Componentes Groupware

En secciones anteriores fue presentado un panorama general sobre los sistemas basados en componentes que hacen a los mismos más accesibles a los usuarios finales de modo de permitirles usar sus entornos de trabajo en una forma más flexible. El sistema de componentes groupware que se intenta introducir tiene como objetivo proporcionar la extensibilidad de los frameworks de componentes, pero con el agregado de mayor flexibilidad y funcionalidad: Dejando a un costado la diferencia de que una componente groupware es *per se* colaborativa (mientras que OLE2 y otros frameworks de componentes no lo son), un objetivo principal del diseño y desarrollo del framework DyCE fue hacer al sistema resultante ampliable y hacer las componentes intercambiables, incluso hasta el punto de usar diferentes componentes sobre el mismo contenido de un documento.
En tal ambiente, las distinciones entre las componentes y el nivel para el cual el usuario puede identificar una aplicación se torna mas y más borrosas. Un usuario ya no trabaja mas en una aplicación sola, mas bien esta decidiendo con que herramienta quiere trabajar sobre los documentos.

4.3.1 Esquematización de un sistema basado en Componentes Groupware

La figura 4.2 muestra un esquema general de cómo es un sistema de componentes groupware.

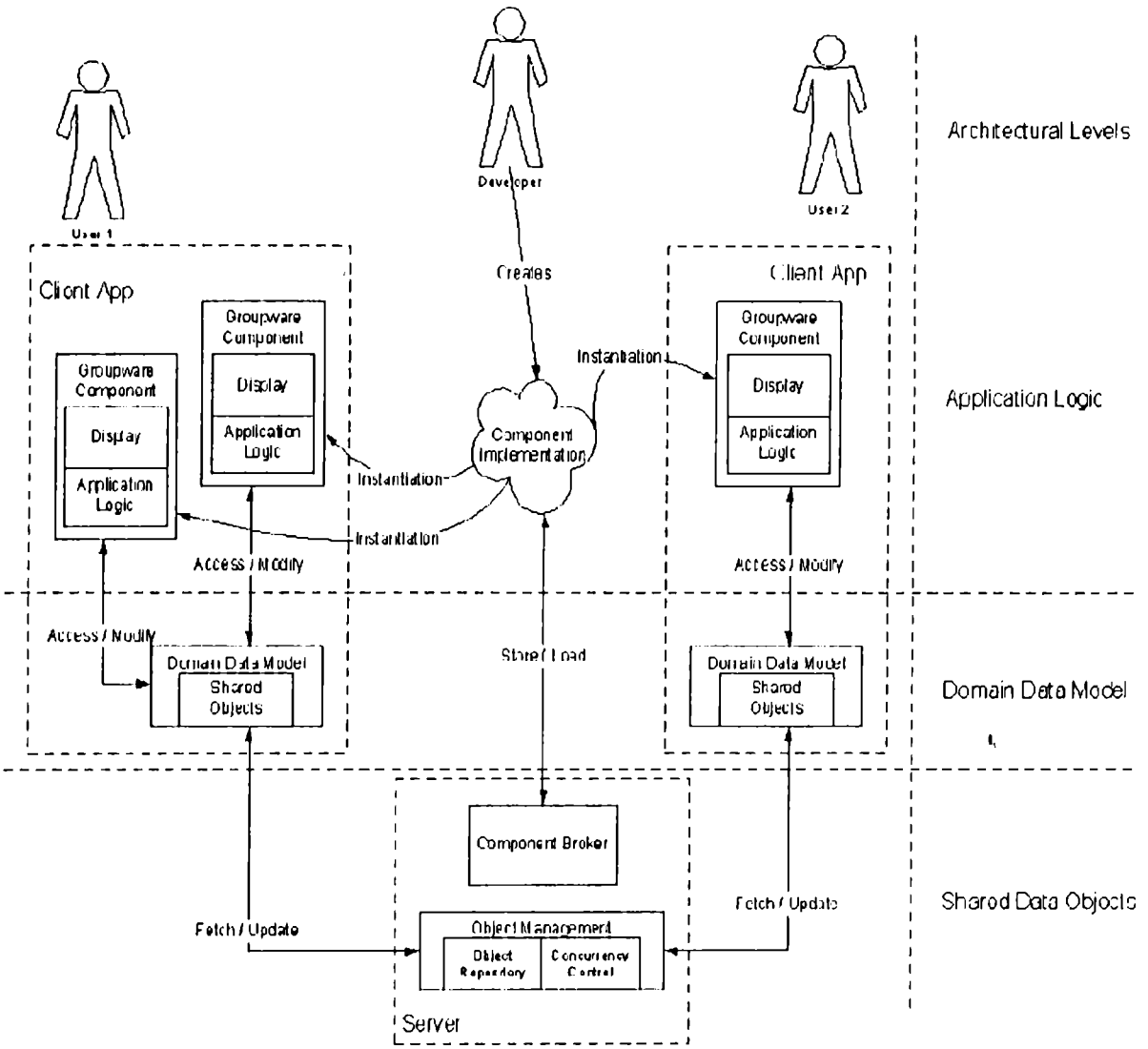


Figura 4.2. Esquema de un sistema de componentes groupware

En el esquema, a cada usuario le es proporcionado una aplicación cliente dentro de la cual los componentes groupware son empleados para acceder y modificar elementos de un modelo de datos en común.

4.3.2 Definición de Componente Groupware

Un *Componente* es un elemento complejo del sistema, que proporciona acceso a los objetos a través de una interfaz de usuario. Conceptualmente, un componente combina una vista (la cual presenta a los elementos de modelo compartido en una forma útil y permite la interacción con dicho modelo), con una implementación de la lógica de la aplicación, la cual es responsable de controlar el acceso al modelo de datos compartido. Los componentes pueden anidarse: un componente puede estar contenido en otro componente y

puede el mismo contener componentes. Adicionalmente, pueden acoplarse varios componentes en uno de los datos compartidos.

Los componentes son implementados por los desarrolladores, quienes crean (programan) las componentes y las ponen a disposición de los usuarios almacenándolos en un subsistema *Component Broker* (algo así como un ejecutor de componentes) accesible por todos. Cuando los usuarios finales acceden al sistema, las implementaciones de los componentes son recuperados desde el Component Broker ubicado en el servidor, luego son instanciados y configurados con relación a los objetos de datos que puedan mostrar y modificar.

Los componentes groupware son accedidos a través de una aplicación cliente, la cual provee a los usuarios de un medio para acceder al ambiente colaborativo e invocar la componente deseada.

4.3.3 Objetos Compartidos

La base para el uso colaborativo de Componentes Groupware esta fundada en el uso de datos compartidos. Como se mostró en diagrama, los componentes acceden a objetos compartidos, los cuales forman la base de la colaboración. Los componentes visualizan el contenido de estos objetos, permitiendo al usuario manipular dichos objetos compartidos y manteniendo una vista consistente con el estado de los elementos compartidos.

Los objetos compartidos representan el estado de la aplicación compartido y común a todos los usuarios que accedan estos objetos. Adicionalmente objetos (no compartidos) pueden ser usados por los componentes, por ejemplo, para representar elementos de servicios solo disponibles localmente o solo usados por un único usuario.

4.3.4 Component Broker

Las implementaciones de los componentes son cargadas desde un Component Broker, el cual administra un repositorio de implementaciones de componentes donde las mismas son almacenadas de forma persistente y disponibles para todos los usuarios conectados.
Pueden agregarse nuevas componentes al Component Broker, de la misma forma que se agregan objetos a una base de datos.

El Component Broker puede ser local a los clientes (por ejemplo, en caso de una aplicación instalada localmente) o, como se muestra en el diagrama, puede ser remoto y accedido vía la red. En DyCE el Component Broker es un objeto centralizado disponible para todos los clientes.

4.3.5 Administración de Objetos

Para poder soportar tanto colaboración sincrónica como asincrónica, los objetos compartidos son almacenados persistentemente en un Repositorio de Objetos, el cual es accedido a través de un subsistema de *Object Management*. El ambiente de la aplicación asegura que los objetos son cargados desde un almacenamiento persistente e instanciados

localmente para ser disponibles a los componentes relacionados que lo requieran. Durante el uso de dichas instancias, los objetos son mantenidos en el Repositorio de Objetos (*Object Repository*).

Mientras los objetos son usados por los componentes groupware, los usuarios están potencialmente cambiando tales objetos. Para permitir la colaboración, es importante que todos los objetos compartidos se mantengan en un estado de consistencia global, es decir, que los cambios hechos a un objeto compartido en un lugar, sean reflejados en todos los otro lugares donde se este compartiendo dicho objeto. Esta es la responsabilidad del Object Management. Este subsistema puede emplear mecanismos de control de la concurrencia para resolver accesos concurrentes a objetos que resulten conflictivos.

4.3.6 Características de los Componentes Groupware

Podemos caracterizar a los componentes groupware como:

- **Colaborativos**; a través del acceso a un modelo de datos compartido, el cual es mantenido en un estado consistente, los diferentes componentes proporcionan funcionalidad colaborativa tanto sincrónica como asincrónica.
- **Interactivos**; los componentes dan al usuario acceso inmediato y cooperativo a los elementos del espacio de objetos compartidos y dan un feedback instantáneo acerca de los resultados de estas operaciones de acceso.
- **Cargados bajo demanda**; en lugar de ser directamente incluidos en las aplicaciones accedidas por los usuarios, los componentes son recuperados bajo demanda desde el repositorio de componentes;
- **Interdependientes**; componentes diferentes (instancias de diferentes clases) pueden acceder objetos de datos en común.
- **Utilizables dinámicamente**; el punto exacto en el tiempo en el cual el usuario invoca una cierta componente no puede ser predeterminado, lo mismo ocurre a la inversa, con el momento en que un componente es descartado.
- **Componibles**; es posible para los desarrolladores de componentes y los usuarios finales crear nuevos componentes groupware por medio de la composición de otros componentes existentes.

Los componentes groupware pueden ser usados en varias formas, dependiendo de los requerimientos del usuario. Entre otros, pueden usarse como:

- Componentes de un escritorio groupware (como es el caso provisto por el desktop de DyCE), proporcionando acceso a diversas funciones, basado en los diferentes elementos del escritorio.
- Elementos colaborativos incluidos en páginas Web (por ejemplo: un indicador de awareness que diga quien esta viendo actualmente la página, incluyendo la habilidad de iniciar una comunicación con tal persona)
- Herramientas colaborativas en dispositivos de mano o móviles

Capítulo 5. DyCE

El desarrollo de Componentes Groupware esta soportado por el framework DyCE (**D**ynamic **C**ollaboration **E**nvironment), el cuál esta totalmente implementado en Java.
DyCE ha ido empleado en numeroso proyectos de investigación, para desarrollar un amplio rango de Componentes Groupware.
DyCE ataca una serie de puntos a tener en cuenta si se quiere desarrollar Componentes Groupware como las descriptas anteriormente:

- Necesita permitir trabajar con objetos compartidos.
- Debe asegurar la consistencia de los datos compartidos.
- Elaborar un mecanismo notificaciones para informar de los cambios que se produzcan en los objetos compartidos.
- Debe emplear un paradigma de programación que vuelva flexible y extensible la incorporación de nuevos componentes groupware al sistema.

5.1 Objetos Compartidos

Los componentes groupware deben proveer acceso compartido a los objetos que forman la base de la colaboración.

Dentro de lo que es el desarrollo de componentes groupware, DyCE provee soporte general para objetos de datos compartidos, soporte para unir el modelo de datos del dominio y soporte para el desarrollo de la lógica de la aplicación.

Definición: Slot
Un slot es un contenedor de datos consistente de un nombre, un tipo y un valor. El término es adoptado de COAST, para distinguir entre los atributos comunes de un objeto (las variables de instancia en Smalltalk, o los fields en Java) y los atributos que forman parte de los elementos que deben ser replicados en tal objeto. El tipo de datos de un slot puede ser cualquier tipo del lenguaje (como por ejemplo String) a los que consideraremos como tipos básicos (no confundir con tipos primitivos como int o boolean), o puede ser una referencia

a un tipo RObject (cuya definición doy a continuación). Un slot provee una interfaz simple para acceder a su contenido y cambiar su valor

5.1.1 Definición: RObject

Un objeto replicable (RObject) consiste de un objeto de tipo básico, el cual es la instancia actual de RObject, y un número de slots, los cuales contienen la parte de datos replicable del RObject. Cada RObject es unívocamente identificable dentro del sistema distribuido por su ObjectID.
Para cada objeto del dominio creado por el programador se le asocia una instancia de RObject que se encarga de contener todos los slots que conforman la información replicable de tal objeto.

5.1.2 Observers de Slot y RObject

Cada slot y RObject pueden ser ligado a u conjunto de observadores (observers), los cuales serán notificados cuando el valor de un slot cambia. De este modo, puede ser implementado cualquier tipo de reacción cada vez que un slot es alterado, como puede ser recalcular o refrescar una vista.. Estos objetos observadores deben implementar la interfaz SlotChangeListener u ObjectChangeListener.

5.2 Domain Model

Se pueden crear estructuras de datos específicas empleando instancias de ROBject y Slot.
La superclase del framework para todos los *domain models*, a partir de donde deben subclasificar los desarrolladores de componentes es ModelObject. Cada instancia de una subclase de ModelObject es relacionada directamente a una instancia de RObject en tiempo de ejecución. Todos los datos compartidos que deba contener la instancia de ModelObject deben ser declarados como slots en el método initializeSlots().
Las subclases de ModelObject pueden conformar estructuras jerarquías, de la misma forma que cualquier otra clase en el modelo orientado a objetos.

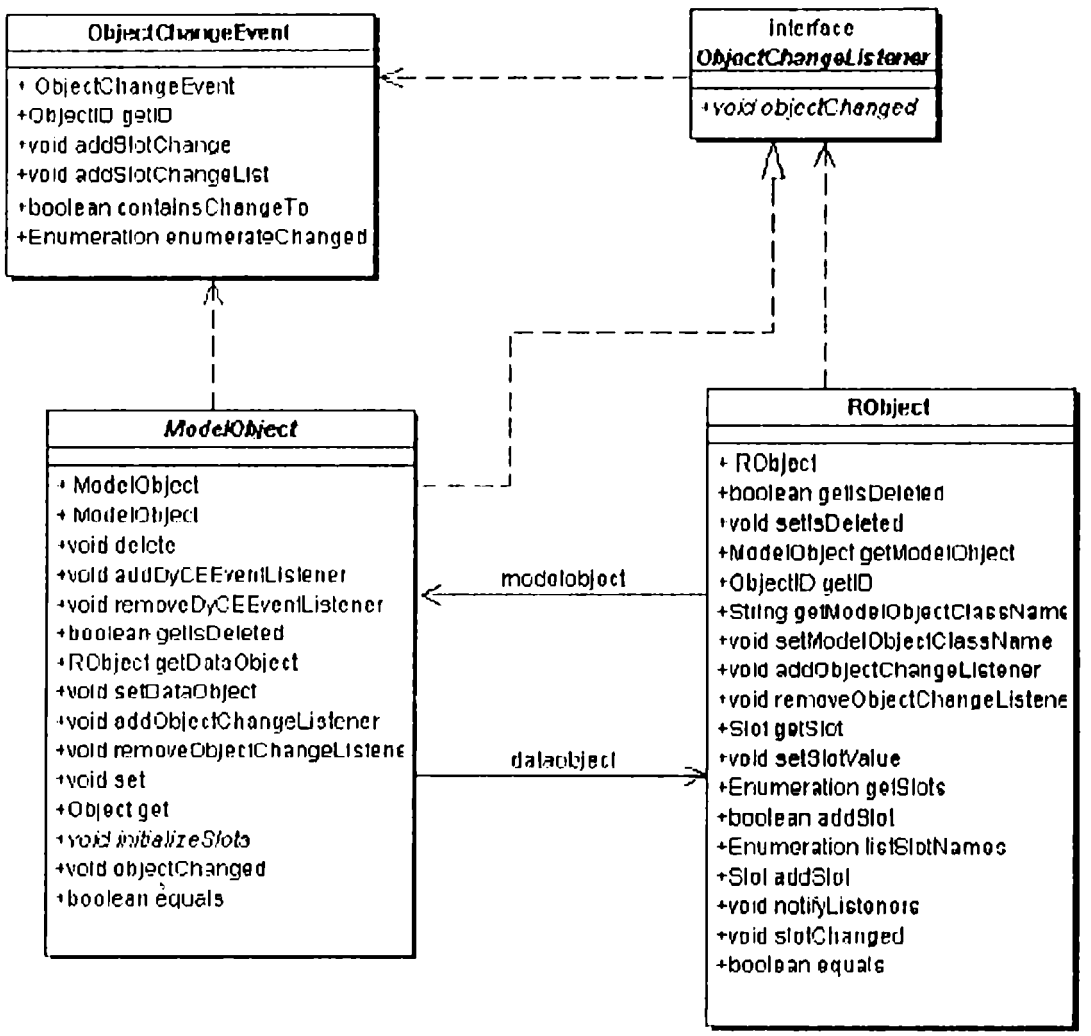


Diagrama 5.1 Soporte DyCE para los objetos compartidos del domain model

5.2.1 Replicación de Objetos

En DyCE se utiliza una variante de lo que se conoce como replicación lazy (véase [LLSG92]), es decir solo se replican los objetos a medida que se precisan, cuando se produce algún cambio en algún objeto compartido, solo se vuelve a traer del servidor el objeto modificado cuando llega a ser referenciado por alguna parte del sistema. En DyCE, inicialmente todos los objetos residen en el servidor y son replicados a los clientes tan pronto como son procesados por éstos. Esta funcionalidad de replicación le concierne al ObjectManager. El sistema de *object managemenet* empleado por el ObjectManager mantiene todos los RObject y maneja su persistencia usando una base de datos orientada a objetos.

La replicación lazy requiere identificar el momento en el cual un objeto es accedidos. Para lo cual el sistema de object management, se compone de dos partes: un ObjectManager residente en el servidor, el cual posee todos los ROBject disponibles, y un ObjectManager

en cada cliente, en donde se mantienen todas las replicas que este empleando cada cliente en particular.

Cuando una componente groupware necesita acceder a un objeto del modelo, emplea el método getObject(ObjectID objID, Object clientID) de ObjectManager. Si el objeto se encuentra disponible en la lista de objetos replicados del ObjectManager cliente, es retornado directamente, de otra forma, deberá ser recuperado desde el ObjectManager servidor y agregado al espacio local de objetos.

5.3 Manejo de Transacciones

La replicación de objetos a través de un sistema distribuido involucra la utilización de un mecanismo que mantenga consistente el estado de tales objetos. Debido a que trabajamos en un sistema cooperativo distribuido que soporta la colaboración sincrónica entre múltiples usuarios, estamos enfrentados al hecho de tener múltiples clientes realizando cambios simultáneos (y por lo tanto, quizás conflictivos) a un mismo objeto, lo cual puede llevar a visualizaciones inconsistentes y a la ruptura de la colaboración.

DyCE emplea Transacciones para manipulación y acceso de grupos de objetos, y utiliza un control de concurrencia centralizado basado en transacciones para mantener la consistencia de los datos. Cada cliente DyCE encapsula las operaciones de acceso a datos compartidos en transacciones.

5.3.1 Tipos de Transacciones

Existen diferentes tipos de transacciones de acuerdo al tipo de operación que se quiera realizar sobre los datos compartidos (lectura, o escritura). Por cuestiones de performance DyCE posee tres tipos de transacciones:

Tabla 5.1. Tipos de Transacciones

Tipo de Transacción	Operaciones permitidas	Descripción
Modify Transaction	Lectura, escritura, creación, cambio	Transacción básica para todos los accesos los datos compartidos. Esta sujeta al control de concurrencia
Display Transaction	Lectura	Para lectura local de grupos, no permite modificación de los datos. No sujeta a concurrencia
Validate Transaction	Lectura, escritura, creación, cambio	Usada solo en el servidor, en la fase de validación y re ejecución de una transacción

Las Modify Transaction son requeridas para cumplir con propiedades conocidas como ACID de los sistemas de administración de bases de datos: Atomicidad, Consistencia, Aislamiento (Isolation) y Estabilidad (Durability) .

Estos términos pueden definirse como:

- Atomicidad, significa que cada transacción se ve como indivisible con respectos a colisiones. En otras palabras, cada transacción ocurre por completo o no ocurre; los efectos parciales no pueden ser vistos.
- Consistencia, significa que cada transacción, cuando es ejecutada sola hasta su culminación, preserva cualquier invariante que haya sido definido sobre el estado del sistema.
- Aislamiento, significa que una transacción es ejecutada como indivisible con respecto a otra: si un grupo de transacciones se ejecutan concurrentemente, el efecto es el mismo que si hubieran sido ejecutadas secuencialmente en algún orden.
- Estabilidad, significa que los efectos de culminar las transacciones (hacer el commit) son lo suficientemente fuertes como para sobrevivir a fallas subsecuentes.

Todas estas propiedades son aseguradas por el Transaction Manager.
La atomicidad es asegurada preservando el valor previo de un slot mientras se efectúa una operación sobre el mismo. Cuando la transacción es completada (committed), el objeto involucrado es escrito en un almacén persistente. Si se produce una colisión, el objeto original puede ser recuperado del almacenamiento sin cambios tal como se encontraba La previo a la transacción.
La consistencia es responsabilidad de los métodos de acceso en las subclases de ModelObject. Los métodos para escribir un slot necesitan chequear restricciones específicas del dominio.
El aislamiento es asegurado por el hecho de que el Transaction Manager cliente no procesará ninguna transacción recibida a través de la red mientras se este ejecutando una transacción local, y viceversa. De esta manera se asegura que la única forma en que puede alterarse el estado de un objeto durante una transacción es por operaciones realizadas en esa transacción.
La propiedad de estabilidad es asegurada gracias al almacenamiento de los RObject en una repositorio persistente. Se asegura que solo RObject consistentes son almacenados, por lo tanto se tiene un punto consistente desde donde poder reasumir la ejecución en caso de problemas.

5.4 Componentes Groupware en DyCE

En la figura 5.2 se puede apreciar el núcleo del framework DyCE, las clases desde donde se pueden realizar las extensiones que permitan el desarrollo de nuevas componentes y las clases para la extensión del framework en sí mismo.

- MobileComponent, la clase base para todos los componentes groupware
- ModelObject, la clase raíz para todas las clases del modelo (domain model)

- RObject y Slot, conforma el modelo de datos genéricos, creados y referenciados desde una subclase específica del dominio de ModelObject.

En secciones previas se explicó la relación entre RObject, Slot, y ModelObject, ahora bien, MobileComponent es la clase a partir de la cual se debe subclasificar cuándo se pretende desarrollar un nuevo componente groupware. Para hacer la nueva componente utilizable en el framework, es necesario implementar los siguientes métodos básicos:
void prepareGUI()
En donde se debe colocar todo el código de inicialización de la GUI (creación de botones, campos de texto, etc.)

void giveTaskBindings(Vector tasks)
Permite agregar las tareas publicadas por este componente (véase la sección5.6 programación basada en tareas).

Cuando se instancia una componente groupware en un cliente, el framework utiliza el método setModel(ModelObject model) de la clase del componente para setear el modelo del component groupware. Luego se invoca el prepareGUI() para configurar la interfaz de usuario de dicho componente.

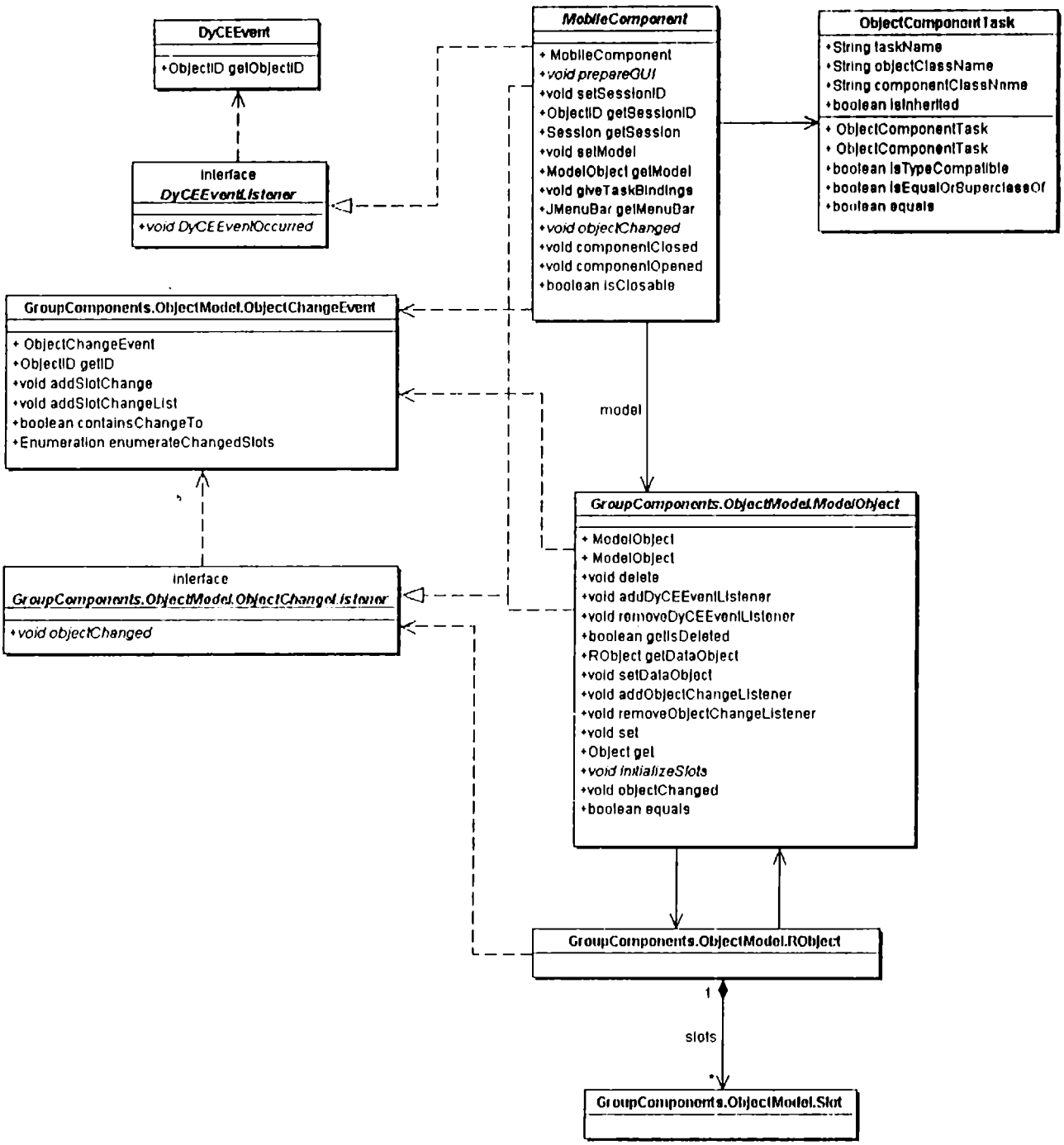


Diagrama 5.2 Componentes en DyCE

5.5 Notificación basada en eventos

En el caso de sistemas distribuidos podemos definir un evento como una ocurrencia especial que toma lugar en cierto punto en el tiempo, aunque no sea reflejado directamente en un cambio de un estado compartido. Estos eventos no solo se relacionan con eventos de la interfaz de usuario (como por ejemplo, el presionar un botón), sino que ocurren también dentro de una componente.

Para ciertas componentes puede ser más sencillo y tal vez más efectivo, el acoplamiento por medio de eventos que realizar complicadas relaciones entre los estados compartidos del objeto para por ejemplo realizar cierta tarea ante un cambio en alguna propiedad compartida. Por ejemplo, supongamos que cierta componente, debe estar atento al login y log out de un usuario de la aplicación, y que ante un login ejecutará cierta función, digamos, reproducir un sonido en particular, y que ante un log out deberá reproducir otro sonido distinto. En este caso, si solo contamos con el compartir datos como primitiva de comunicación, los componentes necesitan estar monitoreando constantemente una lista compartida de usuarios y, cuando ocurre un cambio, comparar los elementos para detectar si un usuario se agregó o si por el contrario alguien se fue, para poder actuar de acuerdo a la situación.

Una alternativa mucho mas simple es un sistema que permita también la comunicación basada en eventos, de este modo, cuando un cambio ocurre, el objeto involucrado emite en broadcast el evento a todos los componentes interesados. Estos componentes interpretarán directamente el evento, de acuerdo a si este es un login o un log out, y de acuerdo a ello sabrá que archivo reproducir.

5.5.1 Extensión de la jerarquía de eventos

La comunicación de eventos entre componentes groupware esta basada en una jerarquía de clases evento. La raíz de esta jerarquía es la clase DyCEEvent. Esta clase puede ser extendida por los desarrolladores para modelar eventos específicos del dominio, por ejemplo en Chatblocks, puede tenerse eventos que modelen la entrada y salida de un participante de la discusión.
La clase base de los componentes groupware, MobileComponent implementa el método DyCEEventOcurred(DyCEEvent event). Este método puede ser sobrescrito por las subclases para poder reaccionar de maneras específicas ante algún evento en particular.

5.5.2 Canales de eventos

Un componente puede hacer un broadcast de un evento a través del método broadcastDyCEEvent(ObjectID, DyCEEvent event). Este método provoca la serialización del evento y la distribución del mismo a todos los clientes conectados que actualmente estén manteniendo una replica del RObject con el ObjectID dado.

Un componente puede registrarse como interesado a un canal de eventos basado en objetos a través de agregarse como DyCEEventListener (es decir implementando esta interfaz) a algún objeto compartido. Esto funciona de forma similar a como es la arquitectura de eventos en Java.

5.6 Programación Basada en Tareas

Cuando se emplean varios componentes para interactuar con objetos del modelo, es necesario mecanismo común para realizar el matching entre un objeto editable y el componente u aplicación utilizada para editarlo.
Hay que tener en cuenta que crear un enlace directo entre el componente y el objeto del modelo es insuficiente ya que en un ambiente colaborativo, usuarios diferentes pueden tener diferentes preferencias acerca de que componente desean utilizar para editar determinado componente (teniendo en cuenta, por supuesto, que haya mas de una).
El mecanismo que propone DyCE es el llamado *task-based programming* (programación basada en tareas)

5.6.1 Definición de Tarea

Podemos definir la relación existente entre un Componente, un Objeto y una Tarea como restringida por las siguientes definiciones:

- Un componente groupware es utilizado para realizar una tarea en un objeto.
- Un componente groupware puede hacer públicas varias tareas diferentes en objetos de clases diferentes.
- Cada tarea tiene un nombre que la identifica
- Para un objeto y tarea dados, la componente groupware requerida es identificada en forma unívoca.

Formalmente, una Tarea es definida por la tripla T = {TaskID, Class, ComponentClass} donde

- TaskID es un ID que le da nombre a la tarea;
- Class identifica a una clase en el sistema;
- ComponentClass identifica a la clase que implementa el componente que hace pública esta Tarea.

Polimorfismo, herencia y sobre-escritura de Tareas

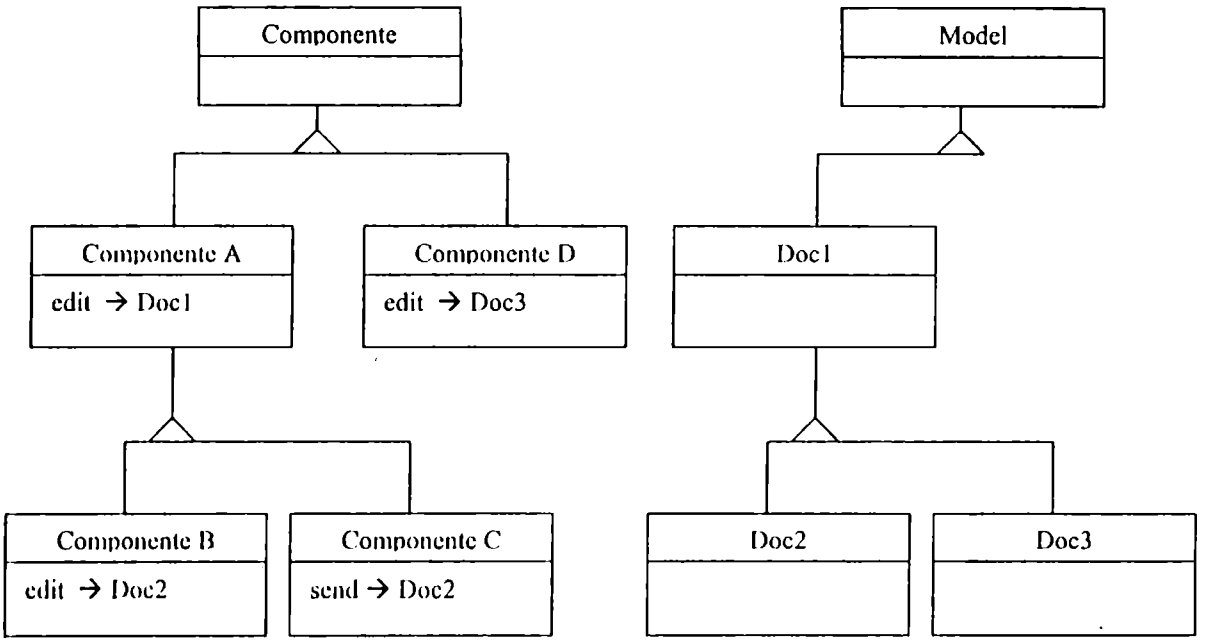


Diagrama 5.3 Ejemplo de herencia de componentes y tareas

Las tareas conforman una estructura jerárquica similar a la estructura de clases de un lenguaje orientado a objetos, con lo que es posible realizar extensiones a dicha jerarquía, o utilizar el polimorfismo o sobre-escritura de Tarea de la misma forma que se haría con objetos comunes. Es decir, dada una componente C1 que publica una tarea T1, si C2 hereda de C1, entonces también habrá una herencia de las tareas, es decir T2 en C2 será una especificación o sobre-escritura de T1.

Algunas definiciones a tener en cuenta en el modelo de tareas son

Tareas publicadas: Se dice que una Tarea es *publicada* por una componente. Que una tarea este publicada en un Componente significa que éste posee la capacidad de realizar cierta operación sobre la clase de objetos declarada en la Tarea. Un componente define cuales van a ser sus componentes publicadas mediante la implementación del método *giveTaskBindings(Vector tasks)*, el componente deberá incluir en el vector task todas las tareas que el mismo publique.

Lookup de componentes: el proceso de Component Lookup recupera el componente definido para una tarea dada sobre una clase de Object dada, es decir retorna un componente del ComponentClass declarado en la tripla que conforma una Tarea.

Ejecución de un Tarea: ejecutar una tarea implica hacer un component lookup que recupera la componente la cual es luego abierta con el objeto declarado en la tarea como modelo.

5.6.2 Tareas como enlaces entre Componentes

La idea central del mecanismo de tareas es que un componente no invoca directamente a otro en su lugar, lo que hará es ejecutar una tarea sobre un objeto del modelo utilizando para ello el sistema de runtime provisto por DyCE, la ejecución de una tarea es la que termina en la invocación del componente correspondiente sobre el objeto-modelo determinado.

5.7 Administración de Sesiones

Una *sesión* es el concepto empleado para modelar a un grupo de usuarios trabajando juntos. Una sesión mapea un conjunto de usuarios en un conjunto de TaskObjectTuple, las cuales relacionan una tarea a un elemento del espacio de objetos compartidos.

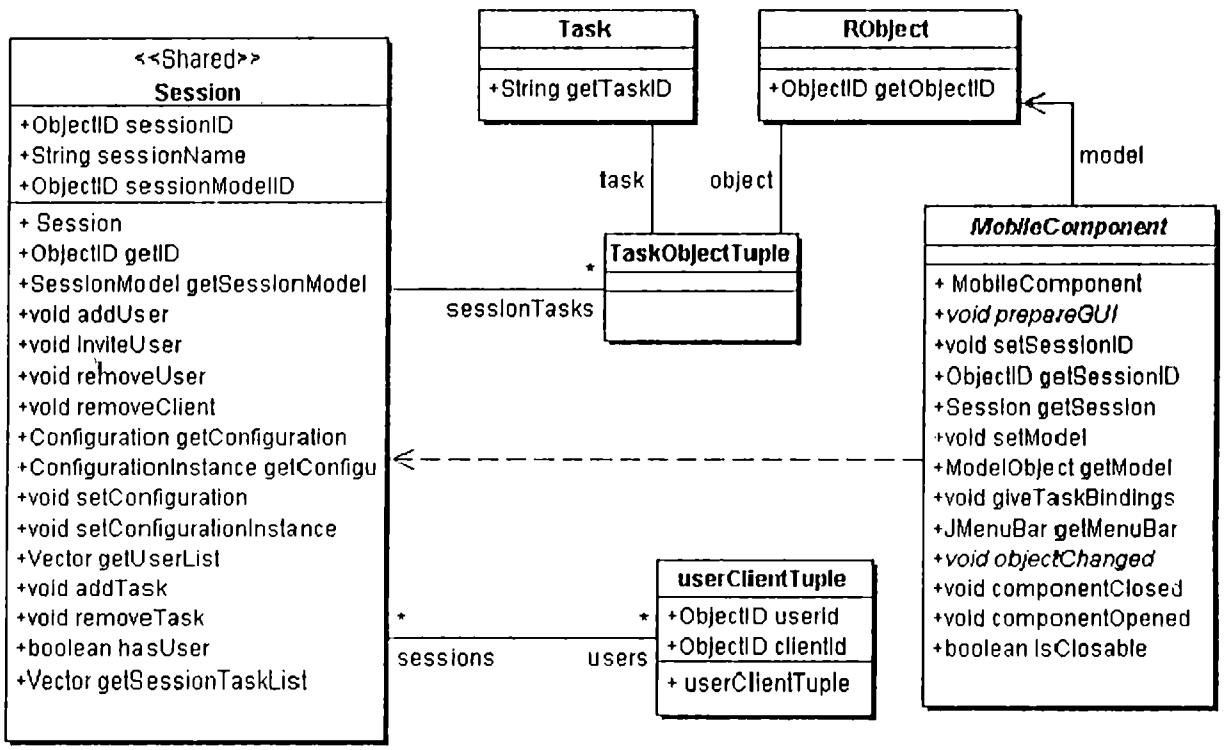


Diagrama 5.4 Modelo de Sesiones de DyCE

Una sesión agrupa las tareas que son ejecutadas sobre instancias de TaskObjectTuple (en realidad sobre los RObject de tal modelo) por todos los usuarios que se hallen actualmente en la sesión. Debido a que el mismo usuario puede estar logeado en múltiples clientes, los

usuarios son representados como tuplas conteniendo no sólo la información del usuario, sino también acerca del cliente al que se refiera en cada caso.
Puede notarse en la figura que la sesión no contiene información directa acerca del componente sino solo los elementos del modelo de tareas. De esta forma, una sesión puede ser suspendida en un sistema (con cierto conjunto de componentes) y puede ser reactivada en otro sistema (tal vez con diferentes componentes, especialmente diseñadas para ese otro sistema, tengamos en cuenta que por ejemplo podría tratarse de un sistema móvil al cual el usuario pretende trasladar el trabajo que estaba realizando por ejemplo en una PC de escritorio).

5.8 Arquitectura del Sistema

De acuerdo a los aspectos de DyCE mencionados hasta aquí podemos resumir la arquitectura conceptual del sistema en el diagrama 5.5. A continuación se dará una breve descripción de capa parte.

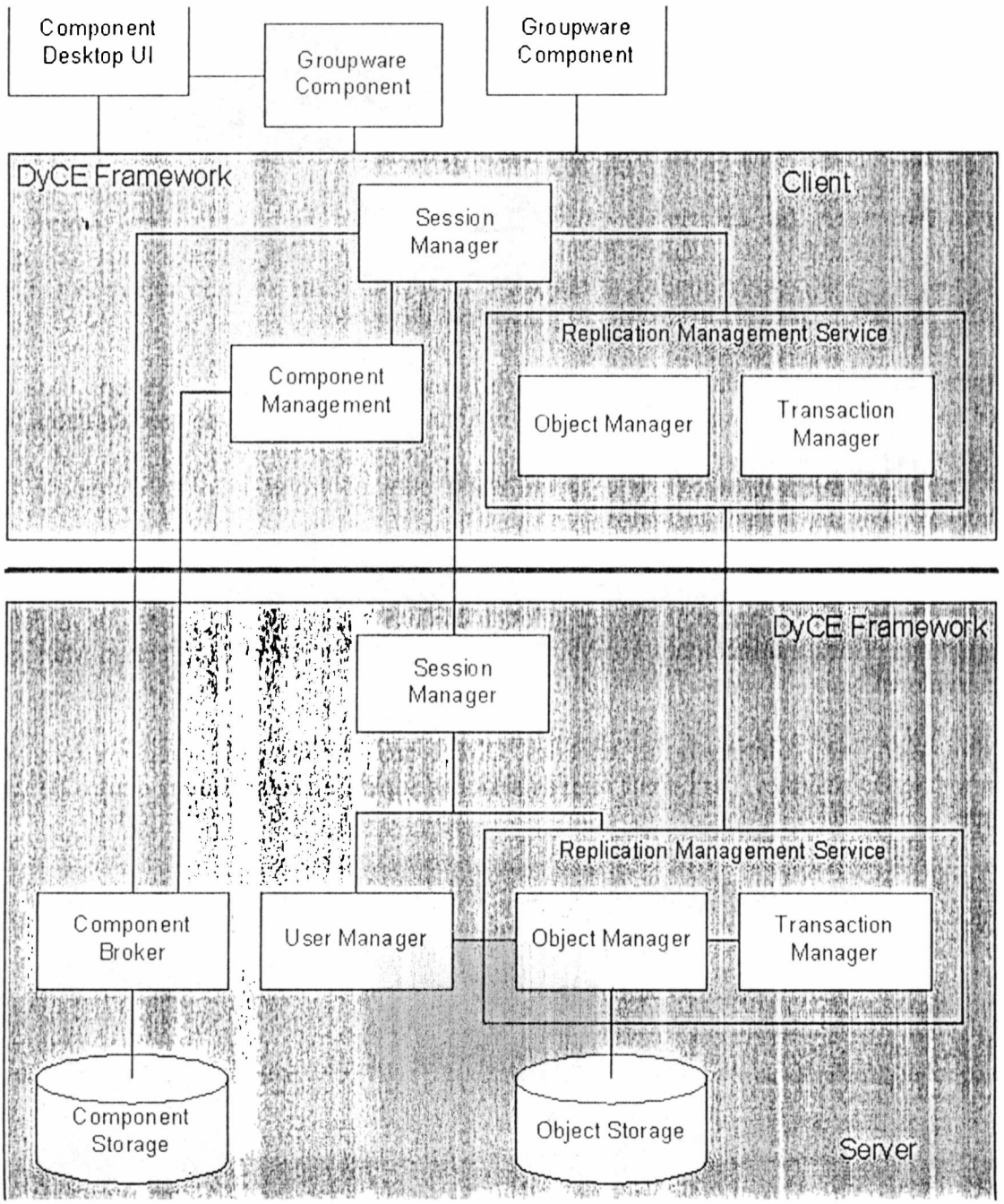


Diagrama 5.5 Arquitectura de DyCE

5.8.1 Arquitectura del Servidor

El servidor consta de los siguientes módulos:

Component Broker: Se utiliza para administrar los componentes disponibles, se encarga de almacenar y recuperar dichas componentes y distribuirlas sobre la red.

Component Storage: Es utilizada para proveer almacenamiento persistente de los componentes. Los componentes son extraídos del Component Storage cuando un cliente lo requiere. El Component Storage almacena los componentes como "*parcels*" ejecutables que pueden ser transmitidos a través de la red y ejecutados en un cliente.

Object Manager: Se encarga de manipular los objetos compartidos en tiempo de ejecución. El servicio del Object Manager es utilizado por los clientes cada vez que necesitan acceder, crear, o modificar objetos compartidos. Este servicio utiliza el Object Storage para manejar el almacenamiento persistente de objetos.

Object Storage: Almacena los objetos compartidos en forma persistente en una base de datos orientada a objetos, su funcionalidad es ejecutada por un sistema de base de datos.

Replication Management Service: Es utilizado para controlar la distribución de las réplicas y la consistencia de los datos.

Transaction Manager: Para asegurar la consistencia de los objetos, el sistema emplea un esquema basado en transacciones: el cliente ejecuta transacciones sobre los objetos compartidos. Tales transacciones son validadas y distribuidas por el Transaction Manager del servidor.

Session Manager: Administra las sesiones. Ofrece una interfaz para agregar y remover usuarios de las sesiones, para crear nuevas sesiones y para ejecutar Tareas dentro de una sesión.

Users Manager: Administra los usuarios registrados en el sistema. Interactúa con el Session Manager para trabajar con la lista de usuarios de las sesiones. Sirve por ejemplo para saber qué usuario esta online en el sistema en cierto momento.

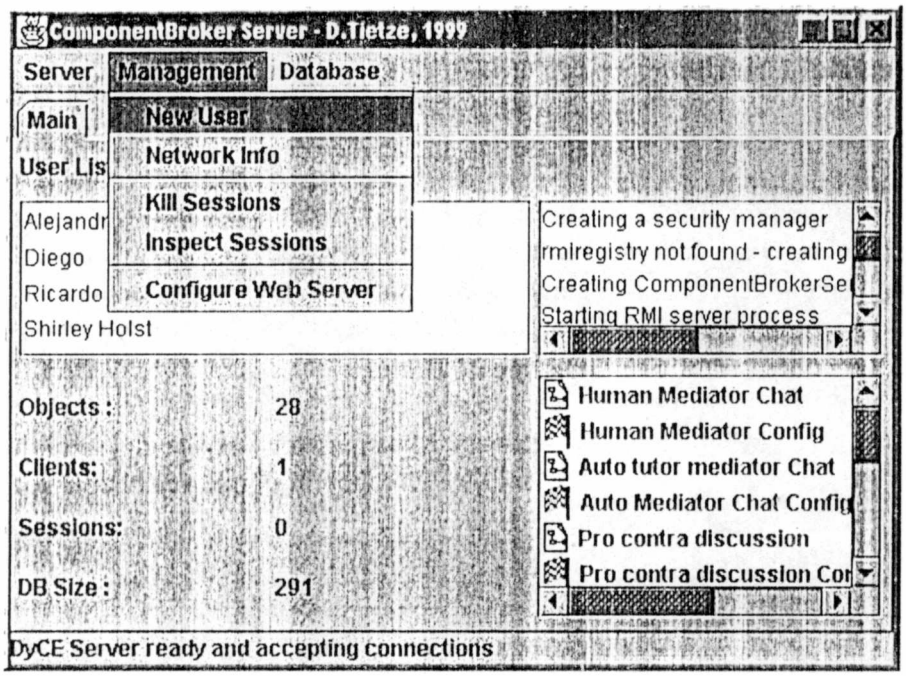


Figura 5.1. Interfaz de Usuario del Servidor DyCE

5.8.2 Arquitectura del Cliente

Cada cliente posee una interfaz, consistente de las instancias de los componentes que proveen la funcionalidad de la aplicación.

DyCE provee un Component Desktop, desde el cual se tiene acceso a todos los componentes disponibles. Este Desktop posee una interfaz que interactúa con el Component Manager para acceder a la información del componente y conseguir la implementación del componente desde el Componente Broker.

El Component Manager es el cliente del Component Management Service del servidor, éste es capaz de averiguar los componentes disponibles, y recuperar el que sea necesario.

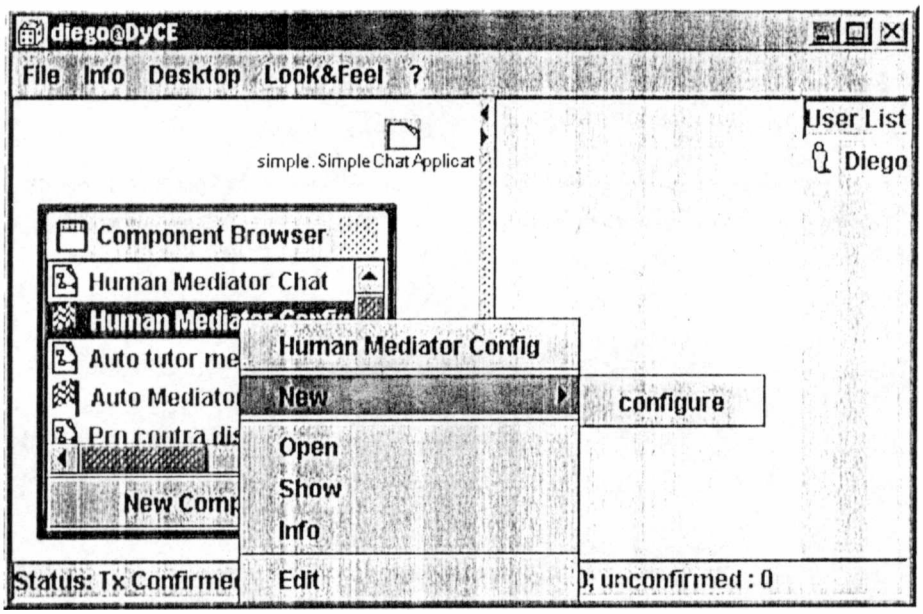


Figura 5.2. Interfaz de usuario del Component Desktop para un cliente DyCE

Capítulo 6. Chatblocks en DyCE

Este capítulo presentará el concepto central de esta tesis, el diseño e implementación de Chatblocks en Java, utilizando la arquitectura DyCE descripta previamente.

El desarrollo de este capítulo esta a su vez divido en dos partes principales.

En la primera, se describirá la filosofía del framework Chatblocks, es la intención es enfatizar la motivación del framework, que tipo de aplicaciones son soportadas por esta implementación, así como también las facilidades que el mismo provee para construirlas. Esta sección es de especial interés para el desarrollador o diseñador de aplicaciones que desee emplear Chatblocks.

La segunda parte, es una guía para el desarrollador, provee una especie de tutoría explicando como es construida una aplicación basada en Chatblocks. Explica como armar una aplicación simple simplemente juntando los bloques necesarios (esta es la finalidad principal de Chatblocks, que el programador logre armar aplicaciones de la misma forma que una interfaz permite generar nuevas componentes visuales a través de la composición de elementos accesibles a través de una paleta de herramientas).

Luego, se dan detalles acerca de cómo construir modelos más complejos, incluyendo protocolos de comunicación más específicos, y describe como utilizar las otras características más avanzadas provistas por esta implementación DyCE de Chatblocks como son el soporte de contribuciones con tipo, o el registro de actividad de la aplicación.

Finalmente, se brindan detalles acerca de cómo armar una interfaz de usuario, y como extender los bloques de componentes visuales provistos por el framework para así crear widgets específicos para un domino en particular.

Primera parte: Filosofía del Framework

Al igual que el prototipo desarrollado en Smalltalk, esta versión final de Chatblocks conforma un framework de caja blanca orientado a objetos. Su dominio específico esta orientado a proporcionar un soporte para la creación de herramientas de comunicación basadas en texto, a estos sistemas se los conoce comúnmente como *chat*.

Chatblocks fue pensado como para tratar a cada una de sus partes como *bloques*, que el usuario (el programador de la aplicación) puede encastrar para componer una aplicación final. Sin embargo, el framework sigue siendo de caja blanca para brindar la mayor flexibilidad posible al programador, quien podrá extender el modelo del framework de la forma que desee simplemente siguiendo las reglas comunes de la programación orientada a objetos (véase el capítulo 2 acerca de los requerimientos del usuario y del desarrollador).

Debido a que una aplicación final creada con Chatblocks conforma un Componente Groupware en DyCE, para evitar confusiones de aquí en mas me referiré a las partes involucradas en una aplicación Chatblocks con el nombre de **bloque**[1]. Así, podría decir, que el framework Chatblocks permite al usuario dividir su aplicación en un conjunto individual de *bloques*; cada uno con una funcionalidad específica que puede ser extendida de la manera que sea conveniente.
Para construir un sistema chat que simule una conversación con restricciones específicas sobre el tipo de aporte que se pueda realizar, el momento en que cada usuario pueda realizar una contribución, etcétera, será suficiente con seleccionar los bloques adecuados que cumplan con tales restricciones (por supuesto tales bloques, pueden contener comportamiento específico de ese dominio de la conversación, para lo cual el programador deberá agregar esa funcionalidad mediante la sub-clasificación de algunos de los bloques básicos provistos por Chatblocks).

[1] Un "bloque local" es una subcomponente Java visual, similar a los componentes Swing. Un "bloque compartido" o "bloque del modelo" es aquel subsistema que forma parte del modelo compartido del sistema, básicamente una clase Java posiblemente sub clase de ModelObject

Como Chatblocks emplea DyCE para manejar la colaboración, y controlar la persistencia y distribución de los objetos compartidos, una aplicación Chatblocks sigue siendo gobernada por las mismas reglas que son empleadas para generar nuevas componentes en DyCE: básicamente se compondrá de un modelo compartido (un ModelObject) y una interfaz local (el componente groupware, subclase de MobileComponent).

Los bloques que componen la estructura general del framework se detallan en la siguiente tabla[2]:

Tabla 6.1. Estructura de una aplicación Chatblocks

Shared Model (ChatApplication)	Local GUI (ChatLocalApplication)
UsersManager	UserActivitiesView
FloorControlManager	FloorControlView
ReceptionManager	ReceptionView
ShipmentManager	ShipmentView
TypeManager	TypeSupportingView (sub bloque de ShipmentView)
RoleManager	Presentado por el ShipmentView
LogManager	Ninguna
MessagePool	Ninguna

En la tabla se puede ver la relación existente entre una clase del View y una clase del modelo. Esto teniendo en cuenta el paradigma Model-View-Controller, si lo pensamos en términos de lo que es una componente groupware en DyCE, los view constituyen los bloques de los que se componen el componente groupware (el MobileComponent), mientras que las clases del modelo, constituyen los bloques que forman el modelo compartido de la aplicación, es decir los ModelObjects.

Cada view es responsable de mostrar una parte específica del modelo global compartido. Por cada bloque abstracto (clase abstracta), el framework aporta al menos un sub-bloque concreto.

6.1.1 Implementación del Modelo de la Aplicación (Shared Application Model)

En el diagrama 6.1 se presenta la arquitectura conceptual del modelo compartido de Chatblocks, por motivos de claridad sólo se muestran las relaciones principales entre cada bloque.

[2] Es importante aclarar que la relación mostrada en la tabla puede no ser siempre 1 a 1. Esto significa que, por ejemplo, el UserActivitiesView puede llegar a mostrar algún estado del FloorControlManager.

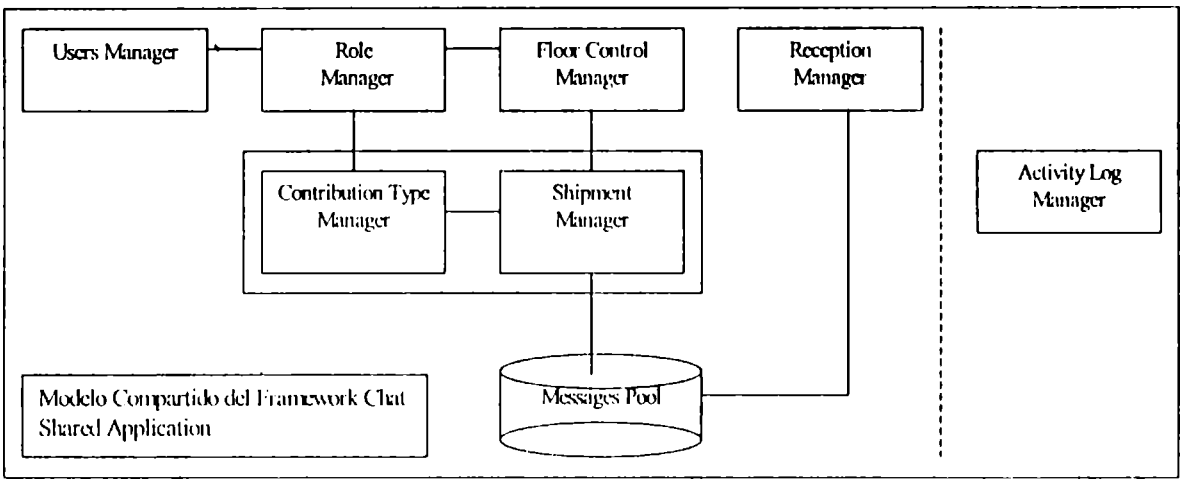


Diagrama 6.1 Modelo Compartido de Chatblocks

El modelo de la aplicación esta compuesto por dos partes:

Un MessagePool, que contiene todos los mensajes enviados durante una sesión, y los Model Managers, cada uno de los cuales interactúa con el MessagePool, por ejemplo, para obtener y mostrar los mensajes enviados, o para depositar en el mismo un nuevo mensaje.

Tabla 6.2. Managers del Modelo

Component	Tipo de clase
UsersManager	Concreta
FloorControlManager	Abstracta
ShipmentManager	Abstracta
TypeManager	Concreta
RoleManager	Abstracta
LogManager	Concreta(*)
ReceptionManager	Concreta

*en la mayoría de los casos no debería ser necesario sub-clasificarla

Algunos de los Managers tales como el UsersManager, el TypeManager, y el LogManager, raramente necesitarán ser sub-clasificados, ya que su funcionalidad es prácticamente independiente del dominio de la aplicación y por lo tanto son bloques que pueden ser empleados directamente en una aplicación final.

Por el otro lado, bloques como el FloorControlManager, el ShipmentManager, y el RoleManager, son específicos para cada aplicación, es decir, aunque el framework provee bloques concretos, solo proporcionan funcionalidad básica.

En pocas palabras se puede describir el funcionamiento de cada Manager como:

Un UsersManager se encarga de llevar un registro de los usuarios conectados a la aplicación, de forma similar a como lo hace el UsersManager de DyCE con todos los clientes conectados.

El FloorControlManager define el protocolo de la conversación, esta íntimamente relacionado con el ShipmentManager, y con los roles involucrados en la aplicación.

El TypeManager administra los posibles tipos de mensajes que pueden (o deben) ser enlazados a un mensaje a ser enviado por el ShipmentManager.

El RoleManager se encarga de organizar los roles de usuario requeridos, si es que la aplicación trabaja con roles.

Por último, el LogManager ejecuta tareas de loging, es decir, va llevando un registro de las acciones o eventos que pueden ocurrir en cualquier parte de la aplicación, por ejemplo el momento en que un usuario entra o sale de sala de chat.

Es importante aclarar, que si bien dije que el usuario es libre de componer sus aplicaciones encastrando los bloques que desee, tal composición no puede ser completamente arbitraria, cada manager espera cierta responsabilidad de los otros, al menos para crear un protocolo de comunicación coherente.

Antes de continuar con una descripción mas profunda acerca de que hace cada manager cabe aclarar algunos términos:

Floor Control

Una vez que un grupo de personas se han juntado en una sesión de conversación, se debe decidir que tipo de acceso se otorgará a cada persona a los elementos compartidos. Por ejemplo, cuando empleamos una pizarra compartida, puede cualquiera dibujar al mismo tiempo (accesos simultáneos), o solo una persona puede acceder a la vez (por medio del pasaje de un token, o turno para utilizarla), o existirá un moderador que dirá quien tiene el control en determinado tiempo, o habrá un limite de tiempo por persona, etc.
Este tipo de situaciones de control sobre los permisos restricciones que se deben imponer en un ambiente colaborativo es lo que se denomina floor control de la aplicación.

En el caso de sistema chat, el floor control estará determinado por el protocolo de comunicación que se quiere modelar. Así, no será lo mismo el floor control que se utilice en una simulación de un debate televisivo (donde por ejemplo el turno es alternado entre los que debaten), y una conferencia de prensa (donde por ejemplo, se podría tener un moderador que diga quien puede preguntar y en que momento).

Rol de Usuario

Dentro de un sistema como el que pretende abarcar Chatblocks, los usuarios pueden jugar diferentes roles, es decir, cumplir determinadas funciones dentro de la conversación, por ejemplo, una persona puede actuar como moderador de un debate, u otra puede ser un alumno en una clase virtual. Estos roles determinan las restricciones y privilegios que tiene cada usuario dentro de la conversación, y las cuales serán regidas por el floor control que se esté utilizando.

Diferentes a los roles son los perfiles de usuarios, estos son inherentes a otorgar a un usuario de privilegios pero del tipo administrativos sobre la aplicación, por ejemplo, que pueda decidir que usuarios serán participantes de una conversación, o poder tener acceso al registro de actividades, o al informe de estadísticas; todas estas son funciones no regidas por el floor control de la discusión.

Contribución

Denominamos contribución al mensaje que un usuario envía al MessagePool.
El tipo de contribución, las características con que puede ser creada, y el momento en que un usuario puede enviarla, son determinadas por el rol del usuario y por el floor control de la conversación.

Awareness

Además de la comunicación explicita, tal como enviar un mensaje o hablar con alguien, muchas situaciones del trabajo en grupo se pueden beneficiar con la comunicación implícita, tal como gestos indirectos, información del entorno (quien está presente, en donde se halla cada uno) o información bibliográfica acerca de la gente que participa de una conversación (como se llama, cual es su posición, o rol dentro de la conversación). Esta información ayuda a la gente a establecer una base común, coordinar sus actividades, y evitar sorpresas.
La información de awareness puede tomar lugar en muchas formas. En una videoconferencia, simplemente proporcionando una cámara de amplio rango, ya se cuenta con un alto grado de awareness. En un cliente de e-mail, simplemente ofreciendo información acerca de la hora y fecha del mensaje o la firma del emisor brinda un contexto acerca del sentido del mensaje.

En cuanto al tipo de awareness necesario en un sistema chat, podemos mencionar: poder conocer quienes están participando de la conversación, que tipo de rol cumple cada uno, quien esta online y quien no, que tipo de actividad esta realizando cada uno, por ejemplo puede ser muy útil contar con algún tipo de awareness que indique si una persona esta componiendo un mensaje o si solo esta esperando a que algún otro lo haga.
Para todo este tipo de cosas, se debe involucrar al UsersManager y al UsersActivityView, dos de los bloques dentro de Chatblocks encargados de proporcionar awareness.

A continuación se dará una explicación concreta de cada una de la clase que componen los bloques del Modelo Compartido de una componente Chatblocks.

ChatApplication

Esta clase encapsula el modelo de la aplicación; es abstracta y conforma la base de todos los modelos de Chatblocks, cada aplicación concreta deberá subclasificar esta clase, y definir los bloques concretos que utilizará.

El framework proporciona una subclase concreta de ChatApplication, SimpleChatApplicationModel, es parte de la aplicación más simple capaz de construirse con Chatblocks, en la que el funcionamiento general es similar al de cualquier sistema chat tradicional, sin protocolos ni roles determinados.

El usuario deberá implementar cada uno de los métodos abstractos que consisten en la creación de los bloques específicos que va a utilizar la aplicación.

El modelo de Chatblocks esta formado por un conjunto de managers, ChatApplication es capaza de crear las relaciones y dependencias entre cada uno de ellos.
El usuario deberá proveer bloques consistentes entre sí a través de la implementación de los siguientes métodos abstractos:

```
public UsersManager createUsersManager()
public FloorControlManager createFloorControlManager()
public RoleManager createRoleManager()
public ShipmentManager createShipmentManager()
public TypeManager createTypeManager()
public ReceptionManager createReceptionManager()
```

En general el LogManager proporcionado por Chatblocks es lo suficientemente configurable como para no necesitar ser sub-clasificado por el usuario, de todos modos si fuera necesario es posible sobrescribir el método

```
public LogManager createLogManager()
```

para proporcionar una especialización más adecuada del mismo.

Dos métodos mas son necesarios implementar para completar el modelo de la aplicación:

```
protected void postModelCreation()
```

El cual sirve para configurar la aplicación una vez que el modelo ha sido creado. Simplemente puede dar una implementación vacía a este método en caso de no necesitar realizar ajustes tras la creación.

El título que se empleará en la vista principal de la aplicación (como título de la aplicación) es proporcionado por el método:

```
protected String createChatTitle();
```

La mayoría de los managers deben ser proporcionado por el usuario[3] , a través de la sub-clasificación de las clases abstractas e implementando los hook methods requeridos por cada uno.

ChatUser

Cada nuevo cliente DyCE que abra un componente Chatblocks dentro de alguna sesión, es encapsulado en un objeto instancia de la clase ChatUser, es decir esta clase encapsula los datos requeridos por el componente Chatblocks, basándose en los datos que la clase User de DyCE proporciona acerca del cliente DyCE. De esta forma es en este objeto ChatUser, donde se concentra la información que permitirá saber si un usuario esta online o no.

La razón por la que se decidió tener un objeto propio dentro del sistema chat que represente al usuario, en lugar de emplear directamente al User provisto por DyCE, es que un usuario que se halle actualmente logeado a una sesión DyCE , no necesariamente será un participante dentro de un componente Chatblocks, o en caso de serlo, el usuario podría estar empleando otros componentes groupware, con lo que para los usuarios de Chatblocks, éste seguiría estando offline (mientras que para DyCE el User pasa a estar online, desde el momento en que se conecta con el servidor).
Además de esta forma, se puede facilitar la extensión del framework, y se gana en la semántica asociada a este objeto.

La clase ChatUser es concreta, y usualmente no es necesaria sub-clasificarla. El UsersManager es un objeto estrechamente ligado con las instancias de ChatUser.

La clase ChatUser brinda acceso de lectura y escritura sobre los siguientes slots[4] de información:

name, es el nombre de usuario dentro de DyCE, el *login name*

icon; una imagen que representa a este usuario dentro del sistema

realname; nombre real del usuario, también obtenido del User de DyCE

logoutTime; almacena el momento en que el usuario se desconecta del sistema Chatblocks (cierra el componente)

---

[3] Hay que tener en cuenta que cada parte del modelo (con algunas excepciones) deben poseer su contraparte en la GUI (es decir en los bloques que conforman el componente groupware), de esta forma, si un modelo utiliza un NoReferenceShipmentMamanger, se debería emplear como view de éste al NoReferenceShipmentView, o alguna subclase de éste, de modo que ambas partes puedan llegan a entenderse correctamente y la aplicación sea consistente

[4] DyCE provee acceso a los slots mediante los métodos get(String slotName) y set(String slotName, Object value). Para facilitar el trabajo con estos atributos compartidos, toda clase que emplea slots brinda una interfaz de getters y setters de los mismos, siguiendo las mismas reglas básicas que se especifican en los JavaBeans para la construcción sintáctica de los mismos. De esta forma, si una clase posee un slot, de carácter público, llamado "slotN", entonces se tendrá acceso al mismo mediante getSlotN() y setSlotN(SlotClassType value)

loginTime; almacena el momento en que el usuario se conecta al sistema Chatblocks (abre el componente)

role; guarda el rol que el usuario cumple dentro del chat (Véase Roles de usuario en secciones anteriores)

currentState ; estado actual del usuario, es controlado por el UsersManager, el mismo puede ser: ACTIVE, AWAY, PINGED, TYPING, OFFLINE

pingString; Almacena un mensaje instantáneo enviado en forma privada por otro usuario (el cual no forma una nueva contribución, ya que no es enviado al MessagePool)

statusTime; se utiliza para decidir cuando el usuario pasa a estar en AWAY, es decir a partir de cuantos minutos es considerado HAWAI.

ChatMessage

Todas las contribuciones hechas durante una sesión de componente Chatblocks son representadas mediante instancias dela clase ChatMessage (o alguna subclase de ésta).

ChatMessage contiene una serie de propiedades que determinan el contenido y semántica de una contribución.
Tales propiedades básicas son:

El **autor** de la contribución, el cual deberá ser una instancia de ChatUser.

Un **timestamp** que indica la fecha y hora de la creación y envío de la contribución.

Una instancia de algún **MessageType** específico (véase Soporte de Tipos, o TypeManager). Esto le aporta mayor semántica al mensaje. Por ejemplo si el mensaje es una pregunta, o si es solo un comentario, o si esta referenciando a algún objeto, el tipo de mensaje puede reflejar a que tipo de objeto referencia.

Y por supuesto el **texto** del mensaje, junto con los atributos del mismo, como son fuente, color y tamaño del texto.

MessagesPool

La instancia de esta clase funciona como un repositorio de mensajes. Esta clase es concreta, y su función es mantener disponibles todos los mensajes enviados durante una conversación.

Cuando un mensaje es enviado al manager instancia de ShipmentManager, éste agrega el mensaje al MessagePool; un ReceptionManager puede leer todos estos mensajes desde el repositorio para poder mostrárselos al usuario a través del ReceptionView.

La clase concreta MessagePool, solo es una abstracción de un almacenamiento de objetos compartidos como puede ser un Vector, simplemente brinda funcionalidad básica para agregar, eliminar y acceder a los mensajes por parte del ReceptionManager. Básicamente se lo puede pensar como una "bandeja de entrada" (inbox) de un programa tradicional de correo.

El MessagePool implementado emplea una clase especial de vector definida exclusivamente para ser capaz de proveer notificaciones más avanzadas que las que puede ofrecer la implementación de Vector que ofrece DyCE. Este tipo especial de Vector, es comentado en detalle en el capítulo 7, donde se presentan algunas extensiones creadas para DyCE, que surgieron durante la implementación de Chatblocks, y cuya función es facilitar la recepción de cambios (que generalmente se deben reflejar en un recalculo en ciertas vistas, o updates).

Como pueden existir restricciones en cuanto a las operaciones que se efectúan sobre el MessagePool, como por ejemplo, la eliminación de un elemento. Puede ser necesario, subclasificar el MessagePool, para capturar este tipo de situaciones, este el caso de la conversación por threads, donde un mensaje que esta siendo referenciado no puede ser eliminado sin antes eliminar sus referencias, de otro modo se provocarían inconsistencias en el contenido del MessagePool. Para este caso en particular, el framework provee una subclase bastante simple de MessagePool, ThreadMessagePool que permite saber cuando es seguro eliminar un ThreadChatMessage.

UsersManager

Es una clase concreta que generalmente no necesita ser extendida, y puede ser incorporada al modelo de un componente Chatblocks sin siquiera necesidad de configuraciones especiales.
Las instancias de este manager se encargan de mantener información referente a los participantes de un sistema chat, se encarga de proveer los ávatars a cada usuario que sea parte de la conversación, es mas puede decidir que usuarios DyCE en particular podrán estar presentes en la lista de usuarios del chat.

El UsersManager, define los perfiles de usuario, es decir, decide quien será el administrador de una sesión, y quienes serán solo participantes.

Cada vez que un usuario abre el componente Chatblocks, el UsersManager recibe el mensaje

```
public void login(ChatUser user)
```

Ante tal evento, el UsersManager agrega al user a su lista de usuarios si es que ya no se encontraba allí, caso contrario lo que hará será cambiar el estado de user a ACTIVE, o provocará una excepción si no es posible, o user no es un usuario habilitado para ingresar a la sesión.

Cada vez que un usuario abandona una sesión, el UsersManager recibe el mensaje

```
public void logout(ChatUser user)
```

En el cual procederá cambiar el estado del usuario user a OFFLINE, y notificar a los interesados.

Un usuario administrador poseerá facultades especiales para decidir que usuarios pueden estar o no en la lista del UsersManager de acuerdo a ciertas características.

ShipmentManager

La clase abstracta base para el subsistema de Shipment Management es el ShipmentManager. Este manager es el responsable de convertir las contribuciones hechas por un usuario en instancias de ChatMessage, o alguna de sus subclases.

Básicamente, su responsabilidad consiste en tomar el mensaje de texto creado por usuario, como un objeto simple instancia de String, crear una instancia de ChatMessage, agregándole las propiedades que sean necesarias, como pueden ser atributos del texto, el tipo, etc.; y finalmente agregar el nuevo mensaje al MessagesPool.

Para cumplir con tales funciones todas las componentes concretas deberán implementar el mensaje

```
public ChatMessage createMessage(String aText,ChatUser cu,
   AttributeForMessage at,MessageType mt);
```

Es decir, de acuerdo al tipo de mensaje empleado, es decir, que clase de ChatMessage utilice el componente, el desarrollador deberá instanciar tal mensaje en este método.

Una instancia de alguna clase concreta de ShipmentManager, puede entender el mensaje
`public void sendMessage(ChatMessage aMesssage)`

La clase abstracta define un comportamiento general para este método que consiste en hacer un chequeo sobre las características del mensaje que se esta intentando enviar:

1. Controla que el usuario este habilitado para enviar el mensaje (restricción que involucra al Floor Control Management)
2. Verifica que el contenido del mensaje sea válido. ChatMessage provee un método por defecto para este tipo de control, que lo único que verifica es que no se esté intentado enviar un mensaje sin contenido de texto (es decir que el texto no sea un String vacío): `public bolean validContent()`
3. Por último debe realizar el chequeo de tipos, es decir que el tipo asignado al mensaje sea válido. Esta acción involucra al ShipmentManager con el Type Manager.

El framework proporciona las siguientes subclases concretas de ShipmenteManager (cada una interactúa con instancias de subclases diferentes de ChatMessage):

NoReferenceShipmentManager, crea instancias de ChatMessage y utilízale método sendMessage() por defecto de ShipmentManager.

ThreadReferenceShipmentManager crea objetos ThreadChatMessage, especialmente apropiados para Threaded communication.

El ShipmentManager recibe pedidos de envío de mensajes por medio del ShipmentView.

Íntimamente relacionado con el tipo de mensajes que pueden ser enviados por el ShipmentView se encuentra el subsistema de Type Management que se explica a continuación.

Soporte para la Clasificación de Contribuciones (Type Management sub-System)

Como una funcionalidad complementaria, se contempla la posibilidad de crear mensajes con tipo, es decir, dado un mensaje ordinario, es posible asignarle un tipo que le aporte un mayor significado (aumentar la semántica de al contribución), por ejemplo para denotar que el mensaje en cuestión es una pregunta, o una referencia a algo, o simplemente un comentario. Con esto, se fortalece el soporte que ofrece el framework para aportar mayor semántica a la información (requerimiento R8) e incluso para brindar un mejor sistema de organización de los mensajes enviados (requerimiento R9).

La clase abstracta encargada de servir como modelo para los tipos posibles es MessageType. El desarrollador puede decidir con que tipos de mensajes desea trabajar, e incluso decidir con que grado de obligatoriedad desea imponer el uso de los tipos a los usuarios (es decir, si la asignación de un tipo determinado a un mensaje será obligatoria u opcional). Para poder tratar con todo este tipo de funciones, el ShipmenteManager, por ser el encargado de la creación de los mensajes deberá contar con la interacción con otro manager: el TypeManager.

A continuación se presentan con mas detalle cada una de las clases de objetos involucradas en el soporte de mensajes con tipo.

MessageType

El la clase abstracta base desde donde se pueden realizar extensiones para derivar todos los tipos posibles para un componente Chatblocks en particular.

Cada una de las subclases deben definir el método:
```
public String getStringRepresentation();
```
El cual define la representación textual para cada tipo, y sirve como identificador global del tipo.

Para poder representar aquellas situaciones en donde el tipo del mensaje no es requerido o no interesa, se provee una subclase especial de MessageType, cuya funcionalidad es completar el hueco que representa trabajar con un mensaje sin tipo: NoTypeMessageType.

Todos las subclases de MessageType, entienden el método
```
public boolean isNoTypeMessageType()
```

Este método por defecto devuelve FALSE; la clase NoTypeMessagetype retorna TRUE, y sirve para manejar los casos especiales en donde debido a la obligatoriedad o no, del uso de tipos pueda requerirlo. Es decir, un ChatMessage que posea como tipo una instancia de NoTypeMessageType pretender significar que tal mensaje no posee tipo.

Desde el punto de vista implementativo, todo mensaje posee tipo, aún cuando la semántica asociada no lo requiera. De este modo se evita en la implementación, tener que tratar con comportamientos distintos ante casos nulos. Esta misma filosofía es aplicable a todo el resto del framework.

TypeManager

TypeManager es una clase concreta, que por lo general no deberías ser extendida para utilizarla. La responsabilidad de esta clase es determinar los tipos posibles de una aplicación para cada rol de usuario, así como también establecer el grado de obligatoriedad en el uso de los tipos.
El TypeManager maneja una lista de con los tipos disponibles, si hay alguno, para cada rol de usuario. Cada vez que el ShipmentManager necesita comprobar la asignación correcta de un tipo a un mensaje, emplea el método

```
public boolean isTypeOk(ChatMessage aMess)
```

Para poder configurar los tipos para cada rol el TypeManager cuenta con los siguientes métodos:

```
public void add(UserRole aur,MessageType mt)
public void remove(UserRole aur,MessageType mt)
public Vector getMessageTypes(UserRole aur)
```

La obligatoriedad en el uso de los tipos puede ser configurada a través del método:

```
public void setTypeSupporting(TypeSupporting ts)
```

El parámetro es una instancia de alguna clase concreta del subsistema de soporte de tipo provistos por Chatblocks:

TypeSupporting

Es una clase abstracta que se encarga de establecer que tipo de exigencias se le impondrán al usuario en cuanto al uso de los tipos en los mensajes. Chatblocks proporciona tres subclases concretas que pretenden cubrir la mayoría de las situaciones posibles cuando se trabaja con tipos:

- **NoTypeSupporting**, es la clase que se debe utilizar cuando se desea construir una aplicación que no desea incluir semántica de tipos a las contribuciones. Es decir, todas tienen igual jerarquía, o categoría dentro de la aplicación.

- **OptionalTypeSupporting**, el uso de esta clase implica que el usuario es libre de elegir cuándo quiere incluir u tipo y cuando no. En términos implementativos significa que el Shipment Manager aceptará mensajes con tipos específicos, pero además tolerará mensajes que contengan como tipo un NoTypeMessageType.

- **ObligatoryTypeSupporting** , si se configura el TypeManager con este objeto, se obliga a que todos los mensajes posean un tipo asignado (es decir, una subclase de MessageType distinta de NoTypeMessageType).

El TypeManager esta compuesto de un TypeSupporting y una Hashtable que mantiene para cada UserRole una colección de los tipos disponibles para tal UserRole. La asociación de los tipos A y B a un rol X, significa que los usuarios con rol X solo podrán realizar contribuciones del tipo A o B, y en caso de que el TypeSupporting sea una instancia de OptionalTypeSupporting, se le da la posibilidad de que no incluya ningún tipo en su mensajes (con lo que el sistema, terminará asignando el tipo NoTypeMessageType a la contribución).

RoleManager

El RoleManager es utilizado para asignar los roles a los usuarios, un usuario puede incluso cambiar de rol durante el transcurso de una conversación, la responsabilidad relativa a qué roles están disponibles y para qué usuarios recae en esta clase. Los roles de usuario son representados en objetos de alguna subclase de UserRole.

Se debe crear una subclase de RoleManager para cada *familia* específica de roles. Una familia de roles es compuesta por una jerarquía de clases con UserRole como raíz (véase UserRole para mas detalles de que es una familia de roles).

La asignación de roles para cada usuario puede ser configurada manualmente por el administrador del componente Chatblocks, por medio de alguna herramienta de configuración, o puede ser configurada automáticamente por la aplicación, de acuerdo al orden en que entren los usuarios al sistema (es decir, se decide otorgar el rol Mediador, por ejemplo, al primer usuario en ingresar el sistema, y el rol Participante al resto).

En caso de que la aplicación no emplee roles, es decir, otorgue las mismas características y habilidades a todos los usuarios, el framework provee la clase NoRoleRoleManager, que solo utiliza la familia de roles compuestas por objetos de la clase NoRoleUserRole.

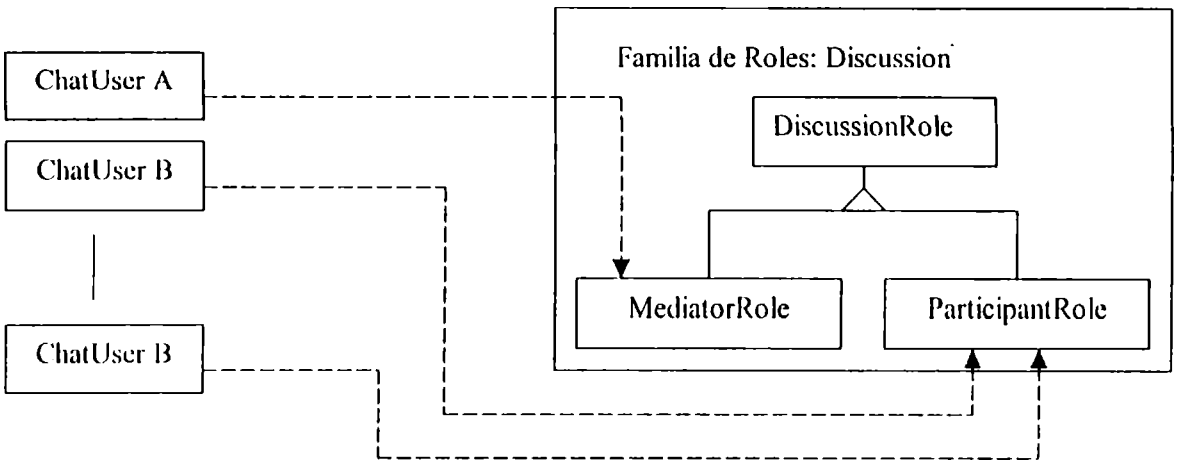


Diagrama 6.2 Ejemplo de un Esquema de Asignación de Roles

UserRole

Es una clase abstracta, sirve como representación del rol que jugará cada participante dependiendo del tipo de conversación que se quiera modelar.
Toda familia de roles debe poseer como abstracción mas general una subclase de UserRole.
Dada una determinada familia de roles, deberá existir un RoleManager específico que se encargue de la asignación de los roles de dicha familia a los ChatUsers que participen de la charla.
En el caso de que se quiera construir un componente que no desee arbitrar de acuerdo a los roles de usuario (es decir, trata a todos los usuarios por igual), el framework proporciona la subclase NoRoleUserRole, cuya semántica es la carencia de un rol específico (es el rol común para todos aquellos participantes que no se diferencian de alguna forma).

Un UserRole no posee estructura, es decir, es utilizado para marcar diferentes tipos de roles sobre los participantes, pero por defecto, no contiene otra información.

UserRole redefine la igualdad de objetos (el método equals(Object o)) de forma que retorna TRUE cada vez que es interrogado con respecto a otro objeto de su misma clase.

UserRoles puede implementarse como un Singleton, pero debido a restricciones impuestas para los objetos compartidos se optó por una implementación regular y se redefinió la igualdad, para simular este tipo de Singleton.

FloorControlManager

Como expliqué anteriormente, el componente Chatblocks, asi como la mayoría de las aplicaciones groupware puede constar de un floor control.

En el caso de un sistema Chatblocks, el floor control es responsable de decidir cuales de los usuarios logueados al sistema esta habilitado para realizar algún tipo de una contribución (enviar un mensaje al MessagePool).
Este tipo de floor control conforma el protocolo de la comunicación. La abstracción sobre la que queda representado dicho protocolo es la clase FloorControlManager.

Cada vez que se quiera modelar un nuevo protocolo de comunicación (es decir un nuevo floor control), se deberá extender la clase abstracta FlooControlManager.

Generalmente, diferentes protocolos, requerirán familia de roles específicas, con lo que involucrarse en la definición de una nueva subclase de FloorControlManager, involucrará también implementar la familia de roles con su respectivo RoleManager.

El FloorControlManager es notificado acerca de los eventos de entrada y salida de usuario, los cuales son atendidos por el UsersManager. Por su lado el ShipmentManager comunica todo envio de mensajes al FloorControlManager.

Cada vez que un usuario intente mandar un mensajes, el ShipmentManager se encarga de verificar la condición de envío necesaria por medio del FloorControlManager, es éste quien decide que usuario esta habilitado para enviar mensajes, y por lo tanto lleva el control de la conversación.

FloorControlManager es una clase abstracta, a partir deben generarse los diferentes protocolos de comunicación de cada conversación en particular.
Las subclases deben implementar el método:

public boolean IsAllowedToWrite(ChatUser aChatUser)
El método recibe como parámetro un usuario del chat, y retorna un bolean indicando si dicho usuario esta autorizado o no para enviar un mensaje. El principal invocador de este método es el ShipmentManager cada que un usuario intenta enviar un mensaje.

La implementación de cómo hace el FloorControlManager para decidir a que usuario se le permite enviar y cual no, queda totalmente librada al diseñador. Generalmente, se puede pensar al protocolo de comunicación como una máquina de transición de estados, donde cada estado es determinado por un UserRole, y las transiciones conducen el flujo de la conversación, cada vez que un usuario de un rol habilitado envía un mensaje.

El flujo de la conversación es alterado por los eventos de entrada y salida de usuario, y por el envio de mensajes, el framework se encarga de hacer llegar estos eventos al FloorControlManager a través de los siguietes métodos:

```
public void login(ChatUser aUser);

public void logout(ChatUser aUser);

public void messageSent(ChatMessage aMessage);
```

Es responsabilidad de las subclases alterar el flujo de la conversación (en caso de tratarse de una maquina de transición de estados, puede pensarse en los métodos anteriores como en disparadores de transiciones), de acuerdo a cada evento recibido.

Al igual que ocurre con el RoleManager, es posible que el componente Chatblocks no desee emplear un floor control management específico, de este modo se comporta un sistema chat tradicional. Para tales situaciones, Chatblocks proporciona un manager cuyo comportamiento es invariante y que siempre habilita a todos los usuarios a enviar mensajes clase NoProtocolFloorControlManager. Este objeto devuelve siempre TRUE en el método isAlloweedToWrite, y simplemente no hace nada en el resto de los métodos disparados por los eventos.

ReceptionManager

Es la clase encargada de la recepción y distribución de los mensajes enviados por los usuarios. Los mensajes enviados son almacenados en el MessagePool, pero el componente groupware no saca los mensajes drectamente desde dicho almacenamiento, sino que debe requerirlos al ReceptionManager.

Si bien, el filtrado u ordenamiento de los mensajes es una funcionalidad local, el propósito del ReceptionManager es agregar un primer nivel general de filtros o captura de mensajes que se ejecuta para todos los usuarios del sistema. Además ciertas funcionalidades generales como son la edición o eliminación de un mensaje del MessagePool, también corren bajo la responsabilidad de este manager. Con lo que si trabajamos con mensajes.

Por ejemplo, supongamos que se le permite a los usuarios eliminar mensajes, y que dicha eliminación necesita de cierta operación sobre el resto de los mensajes debido a que existe cierta relación entre ellos. En este caso, es necesario emplear una subclase de ReceptionManager que lleve a cabo este tipo de tareas.

Soporte para el registro de actividad (Logging Management sub-system)

En esta implementación final de Chatblocks, se incluye como un subsistema especial adicional un administrador del registro de actividad, es decir una herramienta que permite registrar cualquier tipo de actividad sobre la que se esté interesado llevar un control, o para realizar analices posteriores, basándose en el archivo de logging creado por este sub-sistema.

Con este sistema se intenta satisfacer el requerimiento de usuario R12, uno de los que quedaba pendientes en la implementación propotípica de Chatblocks.

El subsistema se basa en la existencia de un manager de carácter general, accesible por todos los otros managers, que registra acciones, ya sean de usuario o de la aplicación. Dichas acciones son abstraídas en objetos de alguna subclase de ActionLog.

LogManager

Es el manager empleado para registrar acciones producidas durante la ejecución del sistema, tales acciones pueden ser eventos de usuario o de la aplicación.

El LogManager va llevando un registro de la acciones, almacenando tal información en un archivo, posiblemente localizado en el sistema de archivos de la máquina que este alojando al servidor DyCE.

Esta funcionalidad puede ser activada o desactivada en cualquier momento durante la ejecución de la aplicación, mediante los métodos:
`public void turnOn()` y `public void turnOff()`

El archivo generado por el LogManager esta en formato de texto, pudiéndose emplear diferentes tipos de estructuras de la información. Por ejemplo puede ser útil que el archivo se construya de acuerdo a las normas de XML, para su posterior procesamiento automático por medio de alguna otra aplicación.

Por ejemplo, si el diseñador desea dejar registrado toda vez que un usuario envia un mensaje, entonces puede incluir en el ShipmentManager una invocación al método

```
public void actionEvent(ActionLog action)
```

con una representación adecuada de ActionLog, la acción será registrada en el archivo log del sistema.

ActionLog

ActionLog es una clase abstracta que sirve como base para todas las acciones registrables por el LogManager.

Básicamente un objeto ActionLog consiste de un ID, el cual es un nombre que servirá para reconocer cada acción en el archivo log. Luego cada subclase en particular poseerá diferentes atributos de acuerdo al tipo de acción que esté representando.

Todas las subclases de ActionLog deberán implementar el método getStrinRepresentation() para darle al LogManager una representación alfanumérica de caracteres empleados para almacenarlos en el archivo log.

El framework proporciona un conjunto de acciones ya definidas que son registradas automáticamente:
StartChatActionLog, la cual es registrada cada vez que el componente chat es abierto.
LoginActionLog, representa el evento que provoca la incorporación de un nuevo participante al sistema.
Y LogoutActionLog, es el evento que se dispara cada vez que un usuario abandona el sistema.

6.1.2 Implementación de la lógica local del sistema e interfaz de usuario (Local Application Model)

La lógica local del componente groupware es encapsulada en un objeto de alguna subclase concreta de ChatLocalApplication.
Esta clase abstracta conforma la base para la construcción de componentes chat.

Un objeto del tipo ChatLocalApplication esta compuesto por un conjunto de views y por el modelo compartido, al igual que cualquier componente tradicional construida con DyCE.

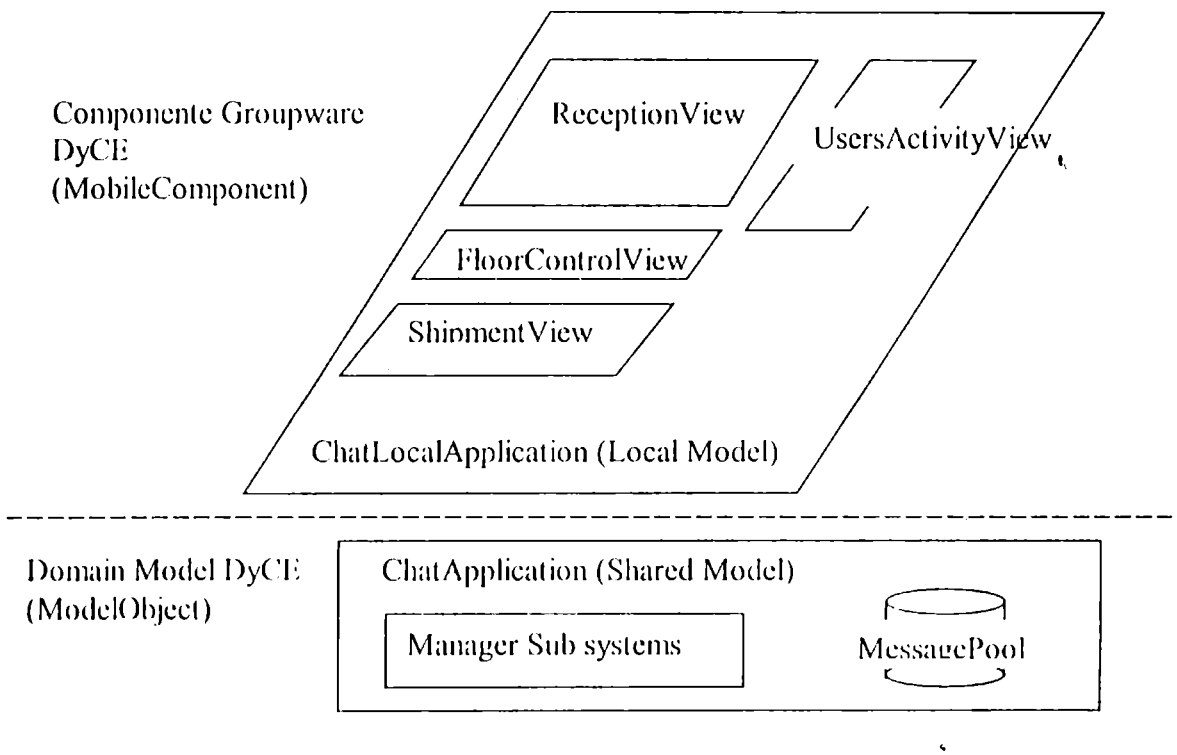


Figura 6.1 Modelo y Componente Groupware de un sistema Chatblocks

Los views son los bloques que contienen la interfaz de usuario del componente y que pueden combinarse entre sí para darle la funcionalidad apropiada al usuario. Cada tipo de view ocupa un lugar asignado por el ChatLocalApplication en la interfaz general del usuario. Chatblocks proporciona, en algunos casos, varios bloques con características algo diferente con relación a como muestran los datos compartidos y a la funcionalidad que le permite al usuario manipular dichos datos.

Tabla 6.3. Los bloques visuales de ChatLocalApplication

View	Tipo de Clase
UsersActivityView	Concreta
FloorControlView	Abstracta
ShipmentView	Abstracta
ReceptionView	Concreta
UsersActivityView	Concreta

Algunos de los views principales mostrados en la tabla de arriba, están a su vez compuestos por otros bloques de menor proporción que pueden ser a su vez diseñados independientemente para obtener mayor flexibilidad.

Tabla 6.4. Los sub-bloques de los Views

Componente	Contenedor
TypeSupportingView	ShipmentView
MessageViewer	ReceptionView
AttributeForTextFieldView	ShipmentView y MessageEditDialog
UserGroupListView	UserActivitieView

Para llevar a cabo operaciones como edición, confirmar una eliminación, mostrar estadísticas, y otras de este tipo, el framework incluye un conjunto de diálogos predefinidos que pueden ser empleados por alguno de los views mencionados.

Tabla 6.5. Diálogos empleados por los Views

Subclase de Frame o Dialog	Bloque Visual involucrado
MessageEditDialog	ReceptionView
MessageExportDialog	ReceptionView
StatisticsView	UsersActivityView

A continuación se presenta un figura que muestra como se relacionan visualmente estos componentes, mostrando de que lugar disponen físicamente:

Figura 6.2 Ejemplo de interfaz de usuario de un componente Chatblocks

Como fue explicado previamente, cada View se corresponde con un manager del modelo compartido.

La forma en que estos Views se disponen y componen para armar una vista como la mostrada en la figura 6.2 queda determinada en la implementación del método createGUI() de ChatLocalApplication.

ChatLocalApplication

ChatLocaApplication constituye la clase base sobre la que deben extenderse los componentes Chatblocks. Toda subclase de ChatLocalApplication forma un nuevo componente groupware que puede ser registrado, por ejemplo en el desktop que proporciona DyCE para tal efecto, de la misma forma que cualquier otro componente.

Un nuevo componente Chatblocks constará de una subclase de ChatLocalApplication (el cual es un MobileComponent de DyCE) y un modelo compartido asociado compuesto por una subclase de ChatApplication (el cual es un ModelObject de DyCE).

Cuando se define un nuevo componente Chatblocks se deben implementar los siguientes métodos:

```
public String getApplicationModelClassName();
```

el cual debe retornar un String con el nombre de la subclase de ChatApplication que compone el modelo del nuevo componente. Este nombre es utilizado por Chatblocks para

crear el binding por defecto entre un MobileComponent y un ModelObject necesario para que DyCE pueda abrir el componente groupware.
También esta parametrizado, en el método *public String getOpenActionCaption()* el nombre de la acción asociada para abrir el componente, por defecto ChatLocalApplication retorna "open" en este método.

Los siguientes métodos deben ser implementados para proporcionar al componente cada uno de los bloques visuales que conforman la interfaz de usuario. Estos bloques serán ligados por el framework con sus correspondientes modelos.

```
public ReceptionView createReceptionView();
public ShipmentView createShipmentView();
public FloorControlView createFloorControlView();
public UsersActivityView createUsersActivityView();
public AttributesForTextFieldView createAttributesForTextFieldView();
```

ChatLocalApplication dispone los views en una forma por defecto, la cual presenta un aspecto como el mostrado en la figura 6.4. Sin embargo las subclases de ChatLocalApplication pueden alterar este orden, e incluso agregar cualquier otro tipo de componente visual, a través de la redefinición del método:
public void chatGUI()

También en ChatLocalApplication se define los métodos que se ejecutan cuando un usuario es cambiado del estado activo a inactivo. Para tal propósito, ChatLocalApplication dispone de un timer que funciona como un indicador de actividad que realiza el usuario, cada vez que el usuario ejecuta alguna acción sobre el componente (aunque más no sea mover el mouse), el timer es reseteado. El valor por defecto fijado para el timer es el contenido en la variable local de ChatLocalApplication *awayTimeinMiliseconds*

UsersActivityView

Es una clase concreta, su principal propósito es brindar información de awareness al usuario, acerca de la actividad llevada a cabo por todos los participantes del chat. El manager del sistema que sirve como modelo de este View es el UsersManager

Este view consiste de una lista donde se exhiben los usuarios registrados en el UsersManager como participantes del sistema.

La lista es capaz de reflejar el estado actual de cada usuario, es decir, por medio de una interfaz, en su defecto icónica, se puede saber si un usuario dado se encuentra conectado, si no se encuentra presente, si esta trabajando en el sistema o su estado es away (es decir, hace un tiempo que no ejecuta ninguna acción sobre el sistema).

Además, es capaz de brindar información estadística de cada usuario, como por ejemplo conocer la cantidad de mensajes aportados por cada usuario, etc.

Para ofrecer este tipo de información el UsersActivityView puede ser configurado con alguna subclase de **StatisticsView**, cuya implementación básica solo contempla el mostrar la cantidad de mensajes aportados por un usuario en particular, desde el inicio de la sesión.

El sistema fue diseñado para ser extendido en sus funciones, por ejemplo podría ser útil incluir una forma de mandar un mensaje prediseñado a un usuario determinado para indicarle que es su turno o que todos están esperando su aporte.

UserGroupListView

Como la aplicación puede emplear diversos estados y roles de usuarios, tales estados y roles pueden verse reflejados en el UsersActivityView por medio de sub-listas que agrupen a usuarios de un mismo rol, o que se encuentran en un mismo estado. Para conseguir este tipo de awareness, el UsersActivity puede ser compuesto por *group list* (que serán objetos de la clase UserGroupListView, o alguna extensión de esta).

Por defecto, un UserGroupListView consta de una etiqueta con el nombre del grupo y una lista donde se muestran los usuarios que pertenecen a ese grupo.
Este tipo de bloques dentro de Chatblocks tiende a mejorar el grado de awareness especificado por el requerimiento de usuario R10.

Los grupos con que puede configurarse el UsersActivityView pueden ser definidos en el método createGUI() de UsersActivityView.

Un UserGroupListView se construye a través de:

```
public UserGroupListView(String title, ListModel lm,UsersActivityView uav
)
```

donde title define la etiqueta que se mostrara como titulo del grupo de usuarios, lm es la lista de usuarios desde donde se obtendrán la información a mostrar y uav es el View donde irá contenido este grupo.

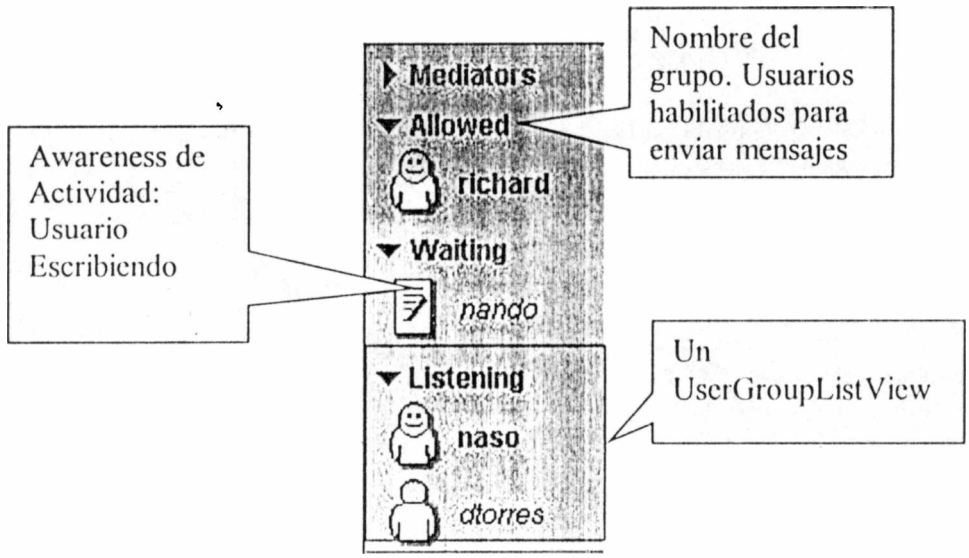


Figura 6.3 Ejemplo de UsersActivityView con varios UsersGroupListView

Chatblocks incluye ciertas clases especialmente diseñadas para poder construir ListModels ordenados por cierto criterio como son ChatUserComparator (empleada para comparar a dos usuarios por algún criterio), y ListFilter que sirve como un wrapper de una lista normal restringiendo la misma a los elementos que cumplan cierta condición. Puede dirigirse a la documentación de las clases de Chatblocks o a las notas sobre extensiones o clases del paquete utilities para mas información acerca de su uso.

ShipmentView

En este bloque del ChatLocalApplication se encuentran los controles de usuario que permiten componer y enviar mensajes.

La interfaz de usuario esta compuesta por un campo de texto, donde se puede escribir el mensaje, y un botón que permite ejecutar la acción de envío del mensaje.

El ShipmentView posee como modelo un objeto de la clase ShipmentManager, y se basa en este para permitir al usuario enviar un mensaje.

Al igual que todos los bloques que forman la interfaz visual de un componente Chatblocks, se pueden redefinir la forma en que se distribuyen los objetos visuales mediante el método createGUI().

El contenido de un ChatMessage es texto plano sin formato, sin embrago el framework permite agregarle cierto formato general pudiendo establecer el tipo de fuente, color y tamaño de la misma. Todos estos atributos son contenidos en un objeto de la clase AttributeForMessage. El ShipmentView puede contener un editor de estas propiedades, el

cual es un objeto de la clase **AttributeForTextFieldView**, este View forma un sub bloque del ShipmentView.

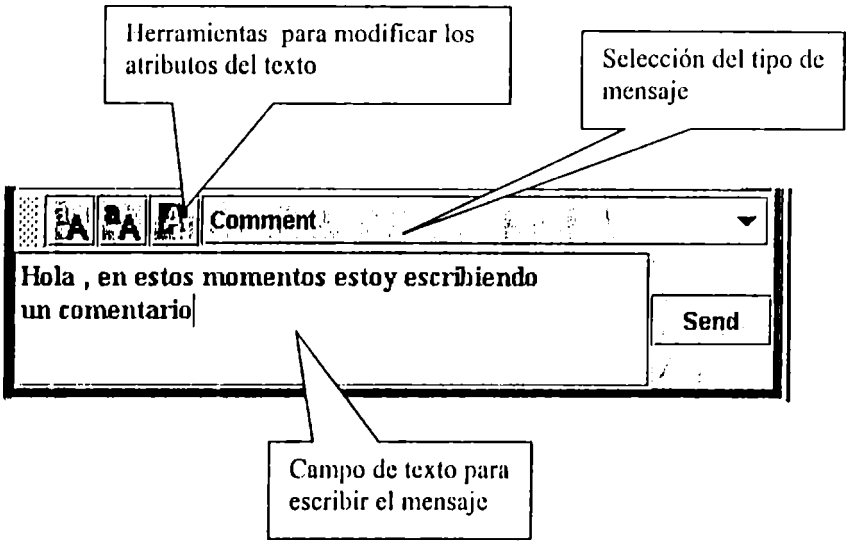


Figura 6.4 Ejemplo de ShipmentView

Como es posible asignar tipos a los mensajes (de acuerdo con el TypeManager empleado en el modelo, en el ShipmentManager), se proporciona otra sub componente visual, **TypeSupportingView**, cuyo aspecto puede variar de acuerdo al TypeSupporting empleado en el TypeManager. Este objeto muestra una lista de los tipos disponibles (o un combo box, en caso de que la asignación de tipos sea obligatoria), o en caso de que la aplicación no trabaje con tipos (es decir, emplee un NoTypeSupporting como modelo), se comporta como una componente no visual (queda oculta al usuario).

El ShipmentView, interactúa con su par el FloorControlView, para decidir cuando habilitar o deshabilitar la funcionalidad de envío (por ejemplo, deshabilitando el botón de envío).

ShipmentView es abstracta, el framework proporciona dos subclases concretas:

NoReferenceShipmentView, el cual envía mensajes ChatMessage simples, sin ningún otro tipo de interacción con otros bloques del sistema.
Generalmente la interfaz de un ShipmentView será la misma para la mayoría de las subclases, ya que esencialmente lo que cambia en este tipo de bloques, es la lógica asociada con el envío.

Como ejemplo de cómo trabajar con mensajes con referencias (en este caso a mensajes previamente enviados), también se ha implementado otro ShipmentView:

ThreadReferenceShipmentView, cuando se envía un mensaje, debe interactuar con el ReceptionView, y de existir algún mensaje seleccionado, el nuevo mensaje es tratado como una respuesta al mensaje seleccionado, agregándose al MessagesPool con una referencia a su padre. En cambio si no existe nada seleccionado, el mensaje se convierte en un nuevo

thread de conversación, almacenándose sin referencia alguna. Puede dirigirse al Anexo referente a implementaciones concretas hechas con Chatblocks para mas información acerca de la implementación de una *threaded communication*.

Cada View esta preparado para trabajar con cierto modelo del dominio, en este caso empleamos en el modelo un NoReferenceShipmentManager, se debería emplear como view un NoReferenceShipmentView (véase la documentación de las clases del framework).

ReceptionView

Instancias de esta clase son las encargadas de mostrar los mensajes enviados por los usuarios.

El ReceptionManager interactúa con el ReceptionManager para obtener la lista de mensajes disponibles para todos los usuarios (recuérdese que el reception manager puede aplicar un primer nivel de filtrado general).

El diseñador puede brindar funciones especiales, como por ejemplo, de edición, eliminación, búsqueda o distintos tipos de visualizaciones extendiendo la clase base ReceptionView.

Si solo se pretende emplear la funcionalidad básica existente en ReceptionView, pero se esta interesado por cambiar el tipo en que los mensajes son visualizados, entonces, no será necesaria la sub-clasificación de ReceptionView, ya que objetos de ésta pueden ser configurados con sub-bloques del tipo MessageViewer. Incluso es posible asignar a un objeto ReceptionView mas de un MessageViewer, y poder intercambiar las visualizaciones en tiempo de ejecución.
Para poder trabajar con estos "visualizadores de mensajes", ReceptionView proporciona los siguientes métodos públicos:

- para agregar un nuevo MessageViewer:

```
public void addMessageViewer(MessageViewer mv)
```

- para eliminar un MessageViewer en particular

```
public void removeMessageViewer(MessageViewer mv)
```

- para decidir que MessageViewer utilizar para visualizar los mensajes en determinado momento se puede emplear:

  ```
  public void showViewer(String name)
  ```

  donde nombre es el ID para tal MessageViewer

ReceptionView provee cuatro operaciones básicas para ejecutar sobre la lista de mensajes:

- Acceder al mensaje seleccionado

```
public ChatMessage getSelectedMessage()
```

- Abrir un diálogo de edición (cuando fuera admitido por el modelo) sobre un mensaje seleccionado

```
public void edit_action(ChatMessage message)
```

- Eliminar (cuando fuera admitido por el modelo) un mensaje seleccionado

```
public void delete_action(ChatMessage message)
```

- Por ultimo, exportar todos los mensajes recibidos a un archivo de texto (por medio del sub bloque ExportView)

```
public void export_action()
```

Por supuesto el programador es libre de alterar o incorporar otra funcionalidad en las subclases de ReceptionView que utilice para crear su componente Chatblocks.

La interfaz MessageViewer

MessageViewer es una interfaz que debe ser implementada por todas aquellas clases que sirvan para crear visualizaciones diferentes de los mensajes recibidos por el ReceptionView.

La forma más simple que brinda Chatblocks para mostrar los mensajes enviados es en una lista, de aspecto similar a un simple JList. Esta visualización es contemplada por la clase ListMessageViewer (que es una extensión de la clase JList de Java).

Precisamente, para evitar ligar la visualización de los mensajes a una componente Swing específica (o imponer la sub-clasificación de una rama en particular de Swing), es que MessageViewer es una interfaz. Basta con definir tres métodos simples para que cualquier componente Swing pueda servir como visualizadora de mensajes:

Para obtener el mensaje seleccionado el ReceptionView empleará el método:

```
public ChatMessage getSelectedMessage();
```

Para poder acceder de forma directa a un MessageViewer en particular es necesario implementar:

```
public String getName();
```

Finalmente, para poder trabajar con objetos vinculados con este MessageViewer, que pueden necesitar de la colaboración con el ReceptionView, se debe implementar el método

```
public ReceptionView getOwner();
```

que devuelve el ReceptionView que contiene a tal MessageViewer

Otra representación visual que ofrece Chatblocks para utilizar como visualizador es la estructura de árbol. Para ello se proporciona una subclase de JTree, TreeMessageViewer, el cual implementa la interfaz MessageViewer.

Los objetos de la clase TreeMessageViewer requieren de un constructor de árboles, objetos del tipo TreeCreator, que se encargan de armar un TreeModel a partir de una lista de mensajes como puede ser un Vector. La ventaja de abstraer en un TreeCreator la forma en que el árbol se construye a partir de una lista, permite una mayor reutilización de componentes de software, ya que aunque cambie el criterio en que el árbol es armado, no es necesario cambiar la representación visual presentada por el TreeMessageViewer (se puede seguir empleando el mismo objeto).

Por defecto un objeto TreeMessageViewer emplea un DefaultTreeCreator como constructor del árbol, este objeto simplemente toma la lista de mensajes y construye un árbol, cuya raíz es nula, y donde todos los mensajes pasan a ser hijos directos de esta raíz.

En cambio, si estamos trabajando con ThreadChatMessage, el atributo parent se vuelve muy útil para dar al TreeMessageViewer un TreeCreator más inteligente, como puede ser el ThreadTreeCreator, que muestre la relación parent-child de forma más natural en un árbol.

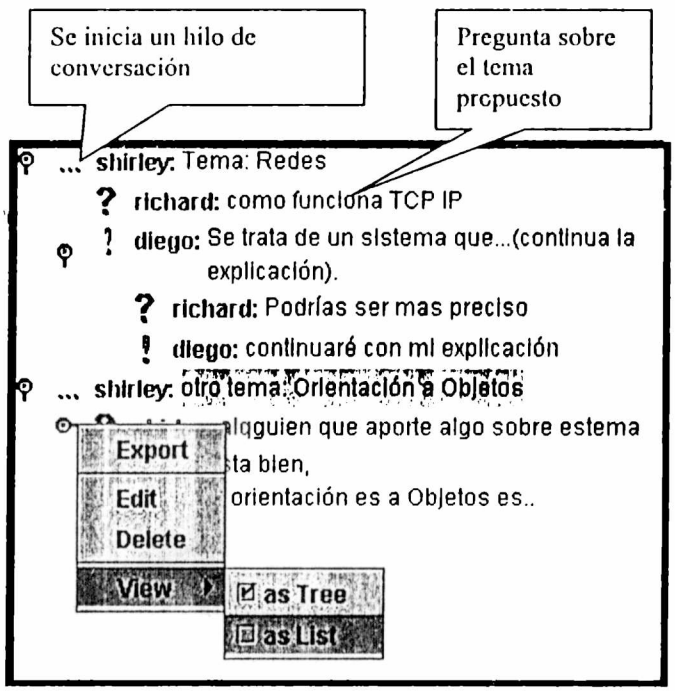


Figura 6.5 Ejemplo de ReceptionView que emplea un ThreadMessageViewer

Objetos de alguna subclase de TreeCreator, son objetos que solo implementan un algoritmo para mapear los elementos de una lista en un árbol.
Un TreeCreator debe implementar los métodos:

```
public TreeNode getRoot();
public TreeNode getNode(Object o);
public TreeNode add(Object o);
public TreeNode remove(Object o);
public Object getObject(TreeNode node);
```

```
public Enumeration elements();
```

Estos métodos son empleados por el TreeModel que construye el TreeMessageViewer a partir de la lista de mensajes y el correspondiente TreeCreator.

Otro aspecto configurable del ReceptionView, es el renderer de los mensajes (es decir el objeto encargado de *dibujar* el mensaje). Además de la forma en que se van a distribuir los ChatMessages en cierta estructura organizativa como son listas o árboles, el programador puede decidir con que forma se mostrará cada objeto particular ChatMessage.
Basta con que cualquier componente Swing sea extendida y e implemente alguna de las interfaces MessageListCellRenderer o MessageTreeCellRenderer, para darle a cada MessageViewer un renderer personalizado.

Por defecto, ListMessageViewer y TreeMessageViewer emplean como renderers objetos de las clases DefaultMessageListCellRenderer y DefaultMessageTreeCellRenderer respectivamente.

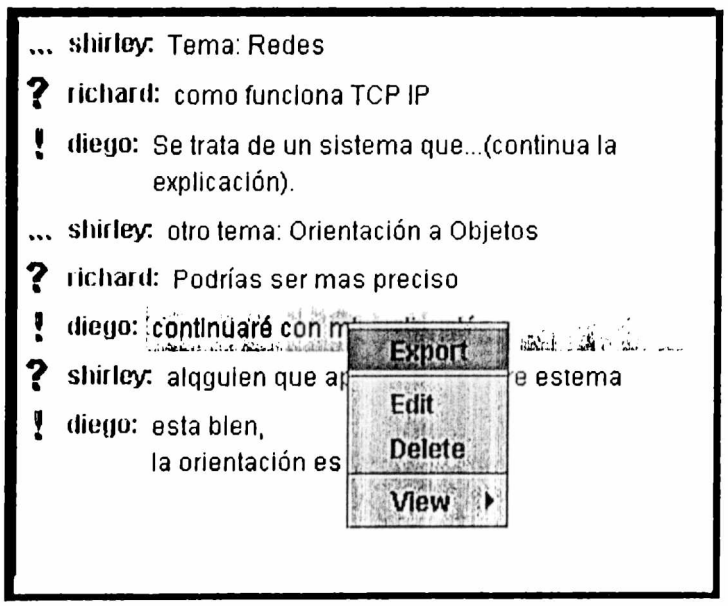


Figura 6.6 Ejemplo de ReceptionView visualizando los mensajes en forma de lista

La flexibilidad que obtiene Chatblocks con todos estos objetos permite satisfacer mejor los requerimientos de organización de la información (R9). Queda totalmente a libertad del diseñador decidir que manera desea mostrar y organizar visualmente la información, para lo que debe subclasificar clases específicas y configura con ellas el ReceptionView. Incluso es posible la configuración en tiempo de ejecución, adaptando cada visualización a los deseos de cada usuario.

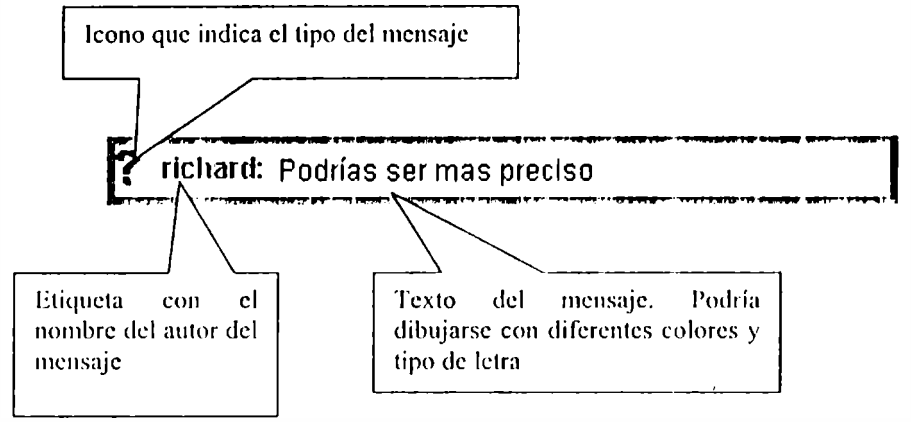


Figura 6.7 Apariencia del DefaultMessageListCellRenderer

La forma de trabajar con renderers es la misma que emplea Java con todos los componentes Swing complejos que puede mostrar modelos mas complicados que un String.

Para satisfacer pedidos de edición o exportación de mensajes ReceptionView utiliza los diálogos MessageEditDialog y MessageExportDialog, los cuales están presentes en el ChatLocalApplication.

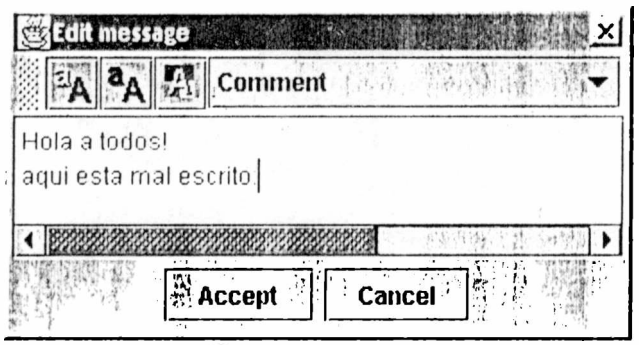
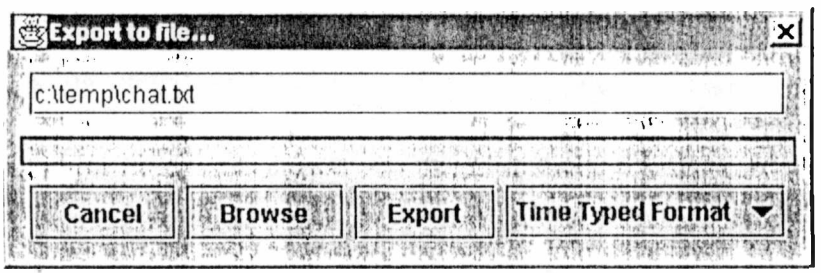


Figura 6.8 Editor de mensajes ya enviados

El MessageEditDialog permite satisfacer el requerimiento R11, referente a la post edición de las contribuciones[5].



[5] Por defecto el framework chequea que el usuario que esta intentando editar un mensaje sea el autor o un usuario con privilegios de administrador.

Figura 6.9 Utilidad de exportación de Mensajes

Con el MessageExportDialog es posible poder guardar toda la información de mensajes enviados en un archivo. Además es posible alterar el formato en que es escrito el archivo, es decir, se podría querer almacenar la información en formato de solo texto, o incluyendo tags, por ejemplo según la norma XML.

FloorControlView

Este boque visual de un componente Chatblocks, muestra información referente al estado actual del sistema, es decir, como se encuentra el flujo de la conversación de acuerdo al protocolo empleado. El FloorControlView puede contener controles para que el usuario ejecute acciones exclusivas de cada protocolo, como por ejemplo pedir la palabra (de la misma forma que un alumno levanta una mano para indicar que tiene una pregunta para hacer en una clase).

Si la aplicación no requiere de controles extras para llevar adelante la conversación virtual, y no es necesario brindar algún tipo de awareness relacionado por ejemplo, a saber, que usuario tiene la palabra en un determinado momento, entonces es posible utilizar como bloque de floor control view un objeto de la clase NoProtocolFloorControlView.

Los objetos de clase NoProtocolFloorControlView, simplemente no tienen una interfaz visual y por lo tanto permanecen ocultos para el usuario.

Por supuesto protocolos de comunicación más complejos y de dominio específico requerirán de un floor control view mas apropiado que contenga labels, mostrando los user roles afectados, o el nombre del usuario que tiene el turno, o cualquier otro tipo de control especial.

Figura 6.10 FloorControl con un único control: un botón de acción (Human Mediated discussion)

Aunque FloorControlView es una clase abstracta, las subclases no necesitan implementar ningún método específico, solo aquellos que requiera para trabajar con el correspondiente floor control manager del componente. En la figura se ve un FloorControlView para una aplicación en la existe la posibilidad de simplemente pedir turnos para hablar (un clic sobre el botón y el personaje se muestra con la mano alzada), o decidir cancelar uno de esos turnos (clic nuevamente sobre el mismo botón para que el personaje baje la mano).

En la figura siguiente se puede ver un FloorControlView que además de incluir un botón para decidir cancelar su turno, presenta cierta información acerca del estado actual del protocolo.

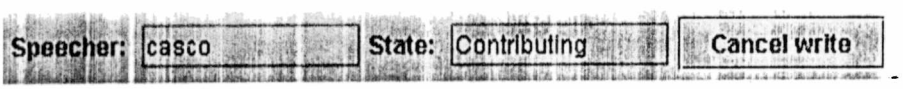


Figura 6.11 FloorControl con información del estado del protocolo (Learning protocol chat)

Segunda parte: Uso de Chatblocks

En esta sección se explicará como construir un componente Chatblocks.
Para comenzar, consideraré el más simple de este tipo de aplicaciones, el cual consiste en un chat simple, para dar una idea de las facilidades que ofrece el framework para desarrollar este tipo de componentes colaborativos. Además el presentar como es construida esta aplicación simple servirá al lector a comprender mejor como es conformada la estructura general del framework y por ende de los componentes que con el se puede construir.

Mas adelante, se mostrarán mas en detalle, como se pueden diseñar y programar componentes groupware más complejas, de modelos mas elaborados, explorando mas las distintas opciones de configuración que ofrece Chatblocks.

6.1.3 Creación de un Chat Simple

Denominaré SimpleChat, a un sistema chat tradicional, de los que se puede encontrar en cualquier sitio web, que consiste en un campo de texto donde el usuario compone y envía sus mensajes (sin ningún tipo de semántica adicional), y en donde todos pueden enviar mensajes en cualquier momento (es decir, sin protocolo explicito de comunicación), y en el cual los mensajes son expuestos al usuario en forma de lista. Para esta aplicación tampoco tendremos en cuenta un mecanismo de tipos.
Las funcionalidades especiales que introduce Chatblocks para enriquecer este tipo de aplicaciones serán explicadas luego, en secciones dedicadas a cada una de ellas.

Como se pudo observar en capítulos anteriores, la aplicación se compone de partes fundamentales: el modelo compartido y la componente que brinda la interfaz de usuario y por lo tanto permite la manipulación de dicho modelo.

6.1.4 Construcción del Modelo de la Aplicación.

El modelo de la aplicación se compone de un domain model formado por un objeto de la clase MessagesPool, y un conjunto de managers.

Todo el modelo de la aplicación (MessagesPool y managers asociados) es encapsulado en una nueva subclase de ChatApplication.

Por lo tanto para empezar habrá que crear una nueva clase a la que llamaré SimpleChatApplicationModel:

```
public class SimpleChatApplicationModel extends ChatApplication{
      /*Constructor necesario para que DyCE asocie el ROBject que
mantendrá la información distribuida de este objeto */
      public SimpleChatApplicationModel(RObject o){
 super(o)
}

/*El constructor habitual empleado para obtener instancias de esta
clase*/
      public SimpleChatApplicationModel(){
 super();
}
```

Ahora bastará con implementar los métodos abstractos definidos en ChatApplication, cada uno de ellos se encarga de configurar el modelo de la aplicación con un manager (o bloque) concreto.

Como ocurrirá con la mayoría de los componentes Chatblocks, se puede emplear como manager de los usuarios del sistema, la clase concreta UsersManager, sin necesidad de introducir modificaciones.:

```
public UsersManager createUsersManager(){
return(new UsersManager());
}
```

Debido a que SimpleChat no requiere de un protocolo explicito de comunicación, no será necesario definir un floor control manager exclusivo para esta aplicación, se puede utilizar como manager para esta función al proporcionado por Chatblocks para este tipo de aplicaciones anárquicas, es decir, el NoProtocolFloorControlManager:

```
public FloorControlManager createFloorControlManager(){
      return (new NoProtocolFloorControlManager());
}
```

Mas aún, como no contamos con un protocolo, tampoco es de interés trabajar con roles específicos de usuario, de modo que podemos emplear como manager de roles un NoRoleRoleManager:

```
public RoleManager createRoleManager(){
```

```
        return (new NoRoleRoleManager());
}
```

Como la aplicación sólo tratará con contribuciones simples, sin ningún tipo de referencias o semántica adicional. El tipo de mensajes que deben ser construidos por el Shipment manager puede ser de la clase ChatMessage. Por lo tanto podemos emplear como manager para el envío de mensajes un objeto de la clase NoReferenceShipmentManager. El método encargado de la configuración de shipment manager queda tan simple como el resto:

```
public ShipmentManager createShipmentManager(){
        return(new NoReferenceShipmentManager());
}
```

El SimpleChat tampoco esta interesado en emplear tipos para clasificar las contribuciones. Sin embargo, para el framework todo mensaje debe tener asociado un tipo (los objetos ChatMessage contienen un slot que indica de que tipo son, *puede referirse a la primera parte de esta documentación para conocer como funciona el soporte de tipos en Chatblocks*)

Como no hay interés en emplear mensajes con tipo, se puede indicar esto utilizando un objeto NoTypeSupportingView como soporte de tipos para el manager de tipos:

```
public TypeManager createTypeManager(){
        return(new TypeManager(new NoTypeSupporting()));
}
```

Finalmente, se debe definir el reception manager, en este caso bastará con el ReceptionManager base, que es una clase concreta en Chatblocks:

```
public ReceptionManager createReceptionManager(){
        return(new ReceptionManager ());
}
```

Al llegar a este punto, ya se han completado todos los pasos necesarios para armar el modelo de SimpleChat. En el futuro, bastará con cambiar alguno de estos bloques por alguno más específico o de funcionalidad distinta para obtener una aplicación de comportamiento diferente. El desarrollador es libre de subclasificar cualquier bloque a partir de alguno de los managers básicos.

6.1.5 Construcción del Componente Groupware

Habiendo definido el modelo de SimpleChat, resta construir el componente groupware, formado por los bloques visuales de que presentan la interfaz de usuario, y proporcionan la lógica local del sistema.

Toda la estructura local así como la interfaz de usuario queda encapsulada en una subclase de ChatLocalApplication. Esta nueva subclase deberá vincularse al modelo, en el caso del SimpleChat, la clase será SimpleChatApplicationModel.

```
public class SimpleChat extends ChatLocalApplication{

public SimpleChat(){
 super();
}
```

Existen dos métodos declarados como abstractos en ChatLocalApplication, y que deben ser implementados para conectar al modelo de la aplicación con el componente que lo manipula:

El método getOpenActionCaption() debe retorna el nombre de la tarea que se empleará para abrir el componente (véase el capitulo de DyCE acerca del modelo de tareas que éste implementa). En este caso, llamaré "open" a la mencionada Tarea:

```
public  String getOpenActionCaption(){
      return "open";
}
```

El otro método, `getApplicationModelClassName` debe retornar el nombre de la clase que encapsula el modelo de la aplicación. Para el SimpleChat debe implementarse de la siguiente forma:

```
public  String getApplicationModelClassName (){
      return "simple.SimpleChatApplicationModel";
}
```

Estos dos String serán empleado en ChatLocalApplication para armar un Tarea con el que sea posible abrir el componente ya sea desde el Desktop DyCE o cualquier otro lugar.

Para definir la interfaz del componente es necesario crear y configurar cada uno de los bloques visuales que toman como modelo a cada uno de los managers definidos en los modelos de la aplicación.
Siguiendo con el ejemplo de SimpleChat, se pueden emplear los bloques visuales que proporciona Chatblocks por defecto para construir una GUI típica:

El Users Activity View queda definido por:

```
public  UsersActivityView createUsersActivityView(){

UsersActivityView uv = new UsersActivityView(
        (UsersManager)this.getUsersManager(),
         SimpleChat.this,
         false);

UserGroupListView ug =new UserGroupListView(
"Users",
```

```
 new SortedListModel(new DDefaultListModel(
  (utilities.ObjectListenerList)
   get(this.getUsersManager().chatParticipantsSlotName))),
   new chatblocks.model.chatUser.ChatUserComparator()),
 uv);

ug.setCellRenderer(new UserListCellRenderer());
uv.addUserGroup(ug);
return uv;
}
```

En este caso, se ha configurado un único grupo de usuarios (un objeto instancia de UserGroupListView), el cual ordenará la lista de usuarios de acuerdo al orden por defecto impuesto por el objeto ChatComparator (define la relación menor o igual entre dos usuarios)

Un objeto de UserGroupListView es creado con tres parámetros: un nombre para el grupo, en este caso "Users", un ListModel, que esta compuesto por al lista ordenada por relación establecida por el ChatComparator, y el UsersActivityView.

SortedListModel, DDefaultListModel, y ObjectListenerList son clases definidas como extensiones para el DyCE, puede encontrar mas información al respecto en el capítulo destinado a mostrar estas utilidades que surgieron como consecuencia de la implementación de Chatblocks y que brindan facilidades para trabajar con listas de objetos compartidos.

Notar que todo bloque visual debe ser construido con el modelo compartido con el que interactuará, en el caso del UsersActivityView es un objeto de la clase UsersManager

ChatApplication define métodos de acceso para cada manager:

```
public FloorControlManager getFloorControlManager()
public ShipmentManager getShipmentManager()
public ReceptionManager getReceptionManager()
public RoleManager getRoleManager()
public UsersManager getUsersManager()
public TypeManager getTypeManager()
public LogManager getLogManager()
```

Por lo tanto desde el componente ChatLocalApplication, se puede acceder a cada uno de ellos mediante:

`getChatAppModel().getX();` donde X es alguno de los managers

Debido a que es necesario dar a cada bloque visual el manager que le sirve de modelo, se puede garantizar la compatibilidad entre el model y el view: si se emplea como modelo un NoProtocolFloorControlManager, entonces no se puede emplear como view de este modelo un ProContraFloorControlView (otro floor control view provisto por Chatblocks) ya que simplemente el constructor de este View requiere otra clase como modelo.

Como modelo del SimpleChat empleamos un NoProtocolFloorControlManager, el view correspondiente es NoProtocolFloorControlView, luego el siguiente método puede emplearse para crear y configurar el bloque visual que se utiliza como vista del floor control de la aplicación:

```
public  FloorControlView createFloorControlView(){
    return new NoProtocolFloorControlView(
   (NoProtocolFloorControlManager)
    getModelApp().getFloorControlManager(),
    this);
    }
```

De la misma forma se puede proceder con los otros dos bloques:

```
public  ShipmentView createShipmentView(){
return new NoReferenceShipmentView(
(NoReferenceShipmentManager)
   getModelApp().getShipmentManager(),
this);
}
```

En este caso se esta empleando el mecanismo por defecto existente en ShipmentView para mostrar los tipos de acuerdo al Type Supporting definido en el manager y a los roles empleados, sin embargo se puede cambiar el TypeSupportingView por defecto empleando el método:

```
public void setTypeSupportingView(TypeSupportingView aTypeSupportingView)
```

Solo resta crear e inicializar el ReceptionView:

```
public  ReceptionView createReceptionView (){
 ReceptionView rv;
 ListMessageViewer mv = new ListMessageViewer(rv=new ReceptionView(
                    this.getReceptionManager(),this ),"List");

 rv.addMessageViewer(mv);

 mv.add(this.getPopupMenu()[1]);
 mv.addMouseListener(this.getMouseListener());
 mv.setListCellRenderer(new PaneMessageListCellRenderer());
 mv.setSelectionModel(new ToggleListSelectionModel());
return rv;
}
```

El ReceptionView es responsable de contener los visores de cada una de las formas en que los usuarios pueden ver y trabajar con los mensajes enviados, a los cuales se accede mediante el ReceptionManager. De esta forma, el programador, puede independizar los tres aspectos del ReceptionView configurables simplemente creando una clase para cada uno de ellos: para cambiar el comportamiento general, como por ejemplo menús, o

[1] getPopupMenu() devuelve un menu apropiado para editar o borrar el mensaje seleccionado, esta funcionalidad puede ser hecha fácilmente por el componente ChatLocalApplication. De igual forma se pueden atender a eventos del mouse desde la misma clase componente, usando el método getMouseListener().

controles extras se puede subclasificar RecpetionView. Si lo que se pretende es tener una forma nueva de visualización de mensajes basta con implementar un nuevo componente Swing que implemente la interfaz MessageViewer. Los MessageViewer proporcionados por Chatblocks (ListMessageViewer y TreeMessageViewer) separan la funcionalidad que muestra los mensajes y los controles sobre las acciones de usuario de la forma en que los mensajes se muestran (es decir, del renderer empleado para dibujarlos), con lo que es posible para el programador emplear uno de estos MessageViewer y cambiar el renderer ajustándolo mas precisamente al tipo de mensaje que se este utilizando (es decir a la clase específica de ChatMessage).

Habiendo definido los métodos que crean y configuran cada bloque visual del componente Chatblocks, solo resta implementar el método que coloca todos estos bloques en el pane principal del componente:

```
public void chatGUI()
```

Este método no es abstracto, ya que por defecto ChatLocalApplication lo define empleando una disposición visual estándar como la mostrada en la figura 6.4

En caso de que el programador desee emplear una disposición diferente o desee agregar otros controles, puede sobrescribir el método chatGUI().

Uno de los trabajos futuros que surgen de esta tesis es construir un entorno visual que permita crear código de forma automática para este tipo de componentes, de la misma forma que se desarrolla cualquier interfaz de usuario con cualquier IDE estándar como el provisto por Jbuilder para aplicaciones Java.

El código por defecto para chatGUI() es :

```
public void chatGUI(){
            int     width       = 630;
            int     height      = 350;
            Font  lbl = new Font("Helvetica", Font.BOLD, 14);
            JPanel vwc = new JPanel();
      VerticalFlowLayout vfl = new VerticalFlowLayout();


            this.setLayout(new BorderLayout());
            vwc.setLayout(vfl);
            vwc.add(this.getReceptionView(), null);
            vwc.add(this.getFloorControlView(), null);
            vwc.add(this.getShipmentView(), null);
            add(vwc, BorderLayout.CENTER);
            add(this.getUsersActivityView(), BorderLayout.EAST);
            this.setPreferredSize(new Dimension(width, height));
      }
```

Con esto se concluye la implementación de un componente de chat simple creado con Chatblocks.

En las secciones siguientes se presentarán las alternativas que tiene el diseñador / programador para cada uno de los bloques que componen un sistema implementado con Chatblocks.

6.1.6 Implementaciones de modelos complejos de conversación

Administración de Usuarios

El UsersManager por lo general no necesitará ser extendido. Sin embargo puede ser necesario cambiar el comportamiento por defecto, por ejemplo para trabajar con mas estados de usuarios que los provistos por defecto. O para registrar otro tipo de acciones, ante la llegada de un evento de login o log out.

El UsersManager recibe los eventos de login y log out de un usuario, ante los cuales cambia el estado actual del usuario y registra la acción como un nuevo ActionLog en el LogManager del sistema. Véase **Soporte para el registro de actividad,** para mas información acerca de cómo opera el subsistema de loging.

El UsersManager emplea dos lista de usuarios, una, conteniendo todos los usuarios registrados en el sistema y que por lo tanto pueden abrir el componente para participar de la sesión. Y otra con los usuarios registrados como administradores, los cuales tienen facultados especiales, por ejemplo para ver o eliminar los mensajes enviados. Si es necesario algún trato diferencial o crear un mayor nivel de seguridad acerca de que usuarios pueden ser registrados se pueden incorporar tales funciones en una subclase de UsersManager.

Extensión de la estructura de las contribuciones

Aunque ChatMessage es una clase concreta, el framework provee otra alternativa, subclase de ChatMessage, que incorpora operaciones adicionales a la clase base: ThreadReferenceMessage:

Como característica determinante en su comportamiento los objetos de esta clase contienen un atributo **parent** que sirve como referencia al padre del mismo. El parent indica una referencia, es decir señala que el mensaje es un sub-tópico de un tema generado por su parent.. La cadena de referencias entre mensaje compone lo que se conoce como threads de conversación. Es particularmente útil para un sistema chat donde se pretende visualizar a los mensajes de acuerdo a esta cadena de threads en forma jerárquica. Para mas información sobre Thread Conversation puede consultar [VC99] y [THDIS97].

Hay que tener en cuenta que si se pretende emplear este tipo de contribución, será necesario definir también otros bloques dentro del sistema para que reflejen las características especiales de este nuevo tipo de mensaje. Comúnmente, los bloques afectados serán ReceptionManager, el ShipmentManager y posiblemente el FlooControlManager.

En el apéndice de esta tesis dedicado a mostrar algunas de las aplicaciones construidas con Chatblocks puede verse una implementación de un thread conversation .

Post-edición de Mensajes

La post- edición de mensaje implica alterar el contenido del MessagesPool, es decir, modificar alguno de mensajes ya enviado por un usuario. El manager que se encarga de este tipo de funciones es el ReceptionManager: por defecto solo permite eliminar una contribución si el usuario que requiere la acción es el mismo que el autor del mensaje.

Para la edición de los mensajes, el ReceptionManager solo se la permite realizar a los usuarios administradores o al autor del mensaje.

Si se pretende redefinir este tipo de comportamiento, adecuarlo a otras políticas o directamente evitar la post-edición se debe subclasificar ReceptionManager y redefinir los métodos involucrados, en especial:
public void deleteMessage()

Uso de Roles

En secciones anteriores se explicó en que consistían los roles y el tipo de familias de roles que pueden definirse para crear grupos de usuarios con ciertos papeles dentro de una conversación o charla.

La clasificación de usuarios que se consigue con los roles es de especial interés en la definición del protocolo de comunicación. El protocolo se basa en distinguir entre roles activos, y no entre usuarios en particular.

Por defecto una aplicación no necesitará emplear roles, esto implica que todos los usuarios son considerados dentro de la misma categoría. Esta clasificación puede obtenerse empleando como role manager un objeto de la clase NoRoleRoleManager.

El role manager es el responsable de asignar un rol a cada usuario. El NoRoleRoleManager lo que hace es asignar el rol NoRoleUserRole a todos los participantes.

Cada nuevo protocolo que utilice roles, debe tener asociado un role manager a cargo de realizar la asignación adecuada de dichos roles.

Suponga que se quiere modelar un protocolo de comunicación para dar una clase virtual, y que dicho protocolo involucra tres roles de usuario: teacher (maestro), student (alumno, estudiante), y listener (solo usuarios que van a oír la charla pero que no tienen participación activa de la misma). Llamaremos a cada clase que representa a estos roles como TeacherRole, StudentRole y ListenerRole.
Es común a agrupar estos roles, que son comunes a un protocolo en particular bajo una superclase que denota la idea de familia de roles y desde donde se sub-clasificarán todos los roles de tal familia. Así podemos definir a esta familia de roles con la clase abstracta ClassRoomRole:

public abstract ClassRoomRole extends UserRole

Una implementación de uno de las tres clases roles puede ser:

```
public class ListenerRole extends UserRole {

  public ListenerRole () {
  }
  public ListenerRole (RObject o) {
          super(o);
  }

/*
Los métodos que siguen son dependientes del protocolo, en este caso se
asume que el protocolo emplea dos roles opuestos y un tercero que solo
escucha las contribuciones de los otros dos
*/

//El protocolo podría emplear este método para decidir si el usuario
pueden realizar una contribución
  public boolean isListener() {
    return true; }
}
```

Uso de Mensajes con Tipo

Considere una herramienta chat que trabaje con contribuciones que pueden estar clasificadas (con algún tipo), es decir, el usuario puede decidir que tipo será su contribución. O la aplicación decide que tipo de mensajes puede enviar un usuario de acuerdo a su rol.

Por ejemplo, puede tener un mensaje del tipo "pregunta", los receptores del mensaje pueden percibir inmediatamente que el emisor hizo una pregunta y que esta esperado una respuesta. Si bien esto no parece de gran ayuda, ya que desde el punto de vista humano, bastaría que el autor del mensaje agregara el símbolo "?" al final para que todos entiendan que es una pregunta (o simplemente por la forma de escribirla), el hacer explícito para la aplicación que ese mensaje es una pregunta le permite al sistema actuar acorde a ello: si el sistema tiene determinado cuales son los roles de usuario habilitados para responder preguntas, entonces puede inhibir a todo el resto de los participantes y solo habilitar a tales roles en particular hasta que alguno dé una respuesta.
El TypeManager es el encargado de definir los tipos disponibles de la aplicación.

Para trabajar con tipos, se debe proveer además de los tipos, un soporte de tipos adecuados (TypeSupporting). El TypeSupporting le permite saber al TypeManager si la especificación de un determinado tipo a un mensaje es **opcional**, **obligatoria** o **no es permitida**.

El framework proporciona una subclase concreta de TypeSupporting llamada NoTypeSupporting para representar aquellas situaciones en donde no se desee utilizar tipos.

Simplemente hay que crear un TypeManager con un type supporting nulo:

```
public TypeManager createTypeManager(){
return new TypeManager(new NoTypeSupporting());
}
```

En este caso, todos los mensajes serán creados con una instancia de NoTypeMessageType como atributo de tipo, los objetos de esta clase indican un tipo vacío o nulo de mensaje.

Suponga ahora que desea incluir mensajes donde la decisión de incluir un tipo es opcional. Esto significa que el usuario es libre de decidir cual de sus mensajes tendrá un tipo asociado y cual no. El type supporting debe ser un objeto de la clase OptionalTypeSupporting:

```
public TypeManager createTypeManager(){
return new TypeManager(new OptionalTypeSupporting());
}
```

Aquí, para los mensajes sin tipo, la aplicación asocia un objeto de la clase NoTypeMessageType, mientras que para aquellos que posean algún tipo en especial, se le asociará el objeto perteneciente a la clase que represente a dicho tipo (otra subclase de MessageType).

La última alternativa es proveer un type supporting que siempre requiera que el usuario especifique un tipo a sus mensajes. En este caso, la clase que asegura este comportamiento es ObligatoryTypeSupporting.

```
public TypeManager createTypeManager(){
return new TypeManager(new ObligatoryTypeSupporting());
}
```

Una vez que se ha decidido con que clase de TypeSupporting trabajará la aplicación, el próximo paso es indicar al TypeManager cuales son los tipos disponibles.

El TypeManager asocia los tipos disponibles para cada rol de usuario.
El método empleado para crear estas asociaciones es

```
public void add(UserRole role, MessageType type)
```

Un rol de usuario puede ser asociado a varios tipos.

Creación de nuevos tipos para los mensajes

Se pueden crear nuevos tipos para asociar a los mensajes mediante la sub-clasificación de la clase MessageType. Cada nueva subclase es tratada como un nuevo y diferente tipo por la aplicación. Lo único que se necesita definir para cada nueva subclase es la representación textual del tipo (un String que debe ser único dentro de los tipos que emplee la aplicación).

Suponga que la aplicación necesita dos tipos de mensajes: preguntas y respuestas.
En este caso se deberán crear dos subclases de MessageType, que podrían ser implementadas del siguiente modo:

```
public class QuestionType extends MessageType {

  public QuestionType () {
            super();
/*se provee un string representativo de este tipo de mensaje */
this.setStringRepresentation("Question");
  }

  /*Debido a que es un modelo compartido se debe proporciona el
constructor necesario para el mecanismo de replicación de DyCE*/
  public QuestionType (RObject o) {
            super (o);
  }
}
```

De esta misma forma podemos definir la clase AnswerType, para los tipos "respuesta".

Habiendo creado las subclases apropiadas de MesssageType, el próximo paso es indicar cuales serán los tipos disponibles para cada rol de usuario.

Supongamos un componente que utiliza tres roles: maestro, estudiante y oyente. Cuyas clases representativas de estos roles podrían ser TeacherRole, StudentRole y ListenerRole, respectivamente.

El TypeManager crea las asociaciones entre roles y tipos mediante el método add(). Una asignación factible para el caso planteado puede ser:

TeacherRole puede enviar mensajes del tipo Answer.
StudentRole puede enviar mensajes del tipo Question.
ListenerRole puede enviar mensajes del tipo Answer y Question.

Para indicar esto al TypeManager, se deben incluir las siguientes líneas de código en la configuración de este bloque del componente (comúnmente en el método createTypeManager() de alguna subclase de ChatApplication):

```
answer = new AnswerType();
question = new QuestionType();

      aTypeManager.add(aTeacherRole, answer);

      aTypeManager.add(aStudentRole, question);

      aTypeManager.add(aListenerRole, answer);
      aTypeManager.add(aListenerRole, question);
```

El ShipmentManager cooperará con el TypeManager para determinar cuando un mensaje posee un tipo correcto, y por lo tanto si es posible enviarlo o no.

Protocolos de Comunicación

El corazón de un componente Chatblocks esta compuesto por el protocolo de comunicación. El floor control manager del componente define dicho protocolo, es decir, la forma en que una comunicación se llevará a cabo. Por ejemplo, en el SimpleChat, el floor control implementa una política irrestricta de comunicación que permite a todos los usuarios escribir y enviar mensajes en cualquier momento y circunstancia. Los usuarios no poseen restricciones, y tampoco es posible distinguir entre diferentes roles de usuario.

Para construir un protocolo de comunicación más específico se debe tener en cuenta los siguientes bloques de Chatblocks:

El UserRole, el FloorControlManager, el TypeManager y el ShipmentManager.

Si deseamos un protocolo que trabaja con diferentes tipos de usuario, se deberá construir una familia de UserRole apropiada y crear una nueva subclase de RoleManager que sea capaz de trabajar dicha familia. Puede observar la sección previa referente Uso de Roles para mas información al respecto.

El FloorControlManager provee métodos para determinar que usuario esta habilitado para enviar un mensaje en un estado (momento) concreto de la aplicación. Este manager tomará decisiones basado en ciertos criterios como ser el rol, el tipo de mensaje, etc.

Para definir un protocolo, se debe sub-clasificar FloorControlManager e implementar los siguientes métodos:

El método principal es

```
public bolean isAllowedToWrite(ChatUser user)
```

que devuelve true si el usuario user esta habilitado para enviar un mensaje.

El FloorControlManager recibe notificaciones de las acciones en el componente que pueden llegar a afectar el curso de la conversación. Estos eventos son notificados con los siguientes mensajes enviados al FloorControlManager:

Un usuario entra al sistema:

```
public void login(ChatUser aUser);
```

Un usuario abandona el sistema

```
public void logout(ChatUser aUser);
```

Un nuevo mensaje ha sido enviado (El floor control puede saber que usuario lo envió mediante el método getAuthor() al objeto aMessage)

```
public void messageSent(ChatMessage aMessage);
```

Estos tres mensajes pueden llegar a afectar el estado actual del protocolo de comunicación.

El protocolo puede ser visto como una máquina de estados, donde cada estado reacciona diferente a un mismo evento.
Cada FloorControlManager definirá otros métodos específicos del protocolo que intenta modelar, y que serán para uso interno, y para conectarse con la vista específica de dicho floor control (una subclase de FloorControlView).

En algunos casos es necesario mantener un estado de usuario dependiendo del estado del protocolo, este estado del usuario es solo para propósitos internos del FloorControlManager, para los otros bloques, la única información relevante será si un usuario puede enviar o no un mensaje. Esta información es obtenida por cualquier bloque invocando el método isAllowedToWrite().

Registro de Actividad (loging)

El bloque LogManager permite al componente Chatblocks capturar eventos y mantener un archivo de registro de los mismos (un *log* del sistema). Vea la sección LogManager para una breve explicación acerca del LogManager y los objetos relacionados.

Todos los componentes creados con Chatblocks poseen una instancia de LogManager por defecto, ésta es creada y configurada en la inicialización de la clase ChatApplication. Para definir el nombre del archivo de log debe sobrescribirse el siguiente método:

```
public String getChatLogFileString(){
        return ("/ChatLog.log");
  }
```

El LogManager puede ser accedido desde cualquier bloque del componente en el modelo por medio del mensaje

```
public LogManager getLogManager()
```

De esta forma, cualquier bloque puede ser capaz de grabar una acción en particular en el archivo de log del sistema.

ChatApplication dispara la primera acción registrada por el LogManager, tras haber sido creado y configurado todo el modelo, y que sirve para indicar que una nueva sesión se ha iniciado: StartChatActionLog

El programador puede definir sus propias acciones de loging para algún propósito en especial y disparar el evento en el momento que lo desee enviando el mensaje

```
public void actionEvent(ActionLog event)
```

al LogManager.

Esta actividad de registro que lleva a cabo el LogManager puede ser activada o desactivada mediante los métodos:

```
public void turnOn()
public void turnOff()
```

Por defecto, el LogManager se inicia como activo, si no es necesario el empleo del LogManager, simplemente basta con invocar turnOff() en el método postModelCreation() de la subclase de ChatApplication.

Una familia de eventos-acciones puede ser definida sub-clasificando ActionLog, los métodos que deben ser definidos son

```
public String  getStringRepresentation()
public String getId()
```

Por ejemplo:
Suponga que se desea registrar el evento que se produce cuando un ChatUser envía un mensaje. Se deben crear las siguientes clases y métodos:

```
public class MessageSentActionLog extends ActionLog {
      /** Representa un nombre de usuario*/
      String user;
      /** Representación de la acción */
      final static String id="MESSAGE_SENT";

/*Se puede crear el constructor mas adecuado de acuerdo al tipo de
evento. En este caso necesitamos el nombre del usuario*/
  public LogoutActionLog(String userName) {
          user=userName;
  }

  /** Retorna la representación en forma de texto de la acción*/
  public String getStringRepresentarion() {
          return " user: "+user+" has sent a message";
  }

/** Devuelve un id de esta acción. Los Id deberían ser únicos
dentro de todas las acciones posibles en un componente*/
      public String getId(){
          return id;
      }
}
```

Ahora, habiendo definido la clase MessageSentActionLog, debemos elegir desde donde disparar el evento que provocará el registro de esta acción. En este caso el lugar más conveniente es en el ShipmentView, cuando finalmente logra enviar el mensaje:

```
public void sendMessage(ChatMessage aMess) {
      /*invoca al método de la superclase abstracta*/
      super(aMess);

/*Código de este Shipment Manager en particular*/
      ...

      /*Dispara la acción*/
          this.getLogManager().actionEvent(new
MessageSentActionLog(userName));
      }
```

Y esto es todo, si el LogManager se encuentra activo, entonces líneas similares a las siguientes serán escritas en el archivo[2] de loging:

```
<MESSAGE_SENT>
Fri Apr 20 11:20:20 GMT -03:00 2001 -
user: nnnn has sent a message
</MESSAGE_SENT>
```

6.1.7 Implementación de vistas complejas del modelo

Mostrar Información de los Usuarios

La información que involucra a los ChatUser puede ser obtenida desde el pane UsersActivityView. Este bloque visual es una clase concreta provista por el framework. Su propósito es dar al usuario de la aplicación un feedback acerca del estado del resto de los participantes de la conversación. Este feedback incluye el poder saber si un determinado usuario esta online u offline, y para todos aquellos que figuren online es capaz de indicar el grado de atención del usuario, si se encuentra en presencia activa o simplemente se haya alejado del computador o envuelto en otra aplicación (estado denominado away). Esto es similar a la forma en que operan los servicios de mensajería instantánea como AOL o ICQ.

Una característica adicional de este bloque visual, es que ofrece la posibilidad al usuario de hacer un llamado de atención a otro usuario (producir un "**ping**"). Con esta función, el sistema abre una nueva ventana (acompañada de una señal sonora) en la interfaz del usuario que recibe el llamado, con un mensaje que le informa que usuario requiere su atención.

También es posible desplegar información estadística de la sesión, por ejemplo, para saber la cantidad de mensajes enviados por un usuario clasificados por tipo de mensajes. Para este tipo de funciones, el UsersAcitvityView incluye un sub-bloque de la clase StatisticsView. Los objetos de esta clase calculan los datos basándose en los mensajes recibidos en el MessagesPool. Por defecto el UsersAcitvityView crea y configura un StatisticsView como se puede ver en los siguientes métodos:

[2] Se puede extender y modificar la clase LogManager para que reaccione de modo diferente cuando ocurre un evento, por ejemplo, para que en lugar de escribir un archivo, ejecute otras Tareas o interactue con otros componentes, abriendo herramientas adicionales al usuario, etc.

```
public void  showStatistics(ActionEvent e) {
```

Obtiene el pool de mensajes

```
MessagesPool mp=(MessagesPool)
(new DisplayTransaction(){
  public void txBody(){
      result=
      getChatLocalApp().getChatAppModel().getMessagePool();
      }
 }).doIt().result();
```

Obtiene el type manager

```
TypeManager tm=(TypeManager)
(new DisplayTransaction(){
      public void txBody(){
      result=
            getChatLocalApp().getChatAppModel().getTypeManager();
      }
      }).doIt().result();
```

Luego con estas líneas, se construye y abre un dialogo con el pane StatisticsView:

```
StatisticsView sv=new
StatisticsView((ChatUser)dyceList.getSelectedItem(),mp,tm);
      sv.show();
```

Todo componente Chatblocks debe proveer un UsersActivityView como parte de su GUI. La clase UsersAcitvityView puede ser empleada en la mayoría de las aplicaciones prácticamente sin realizar modificaciones.

La forma en que se crea y configura este objeto es mediante la implementación del siguiente método de la subclase concreta de ChatLocalApplication que conforme el componente Chatblocks:

```
public  UsersActivityView createUsersActivityView(){
  return new UsersActivityView(
 (UsersManager)getModelApp().getUsersManager(),
  this);
      }
```

Como ocurre con todos los bloques de Chatblocks, el programador es libre de subclasificar y personalizar este objeto según crea conveniente.

Configuración y uso de los grupos de usuario. Implementación

El UsersActivityView es una subclase de JList que esta compuesta por una o varias views más pequeñas llamadas "group views", cada group view esta compuesto, por defecto, de un JLabel y un JList , en esta lista interna se presentan todos los usuarios de un mismo grupo.

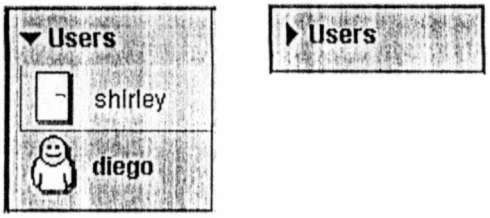


Figura 6.12 Un UserGroupListView básico con la lista de usuarios desplegada (izq.) y contraída (der.)

El propósito de esta disposición es brindar la máxima libertad al programador para personalizar el UsersAcitvityView, creando los group view con los criterios de orden y condiciones que desee.
Como ejemplo de la flexibilidad de esta estructura, puede verse el siguiente fragmento de código: se implementa un view que divide a los usuarios (los filtra) en dos grupos, uno de ellos, es el grupo de usuarios que están habilitados para enviar mensajes, el otro grupo lo forman los usuarios que aún no se les otorgó permiso para enviar mensajes.

Se debe proveer a cada group view del modelo de datos a mostrar, en este caso son ChatUsers: una forma fácil de hacerlo es empleando las clases extras creadas como extensiones simples de DyCE:

DyCEFilteredListModel: se encarga de filtrar los elementos de un ListModel de acuerdo a las condiciones de un objeto ListFilter

DDefaultListModel: adapta un ObjectListenerList a una interfaz ListModel

SortedListModel: representa un ListModel ordenado por medio de un objeto Comparator.

Además de poder mostrar información ordenada o filtrada de los usuarios, también es posible decidir que forma se visualizará dicha información. Es decir es posible cambiarla representación de los usuarios en cada group view, simplemente definiendo el renderer:

```
allowedList.setCellRenderer(new ProContraUserListCellRenderer());
```

Ejemplo:

```
public class SortedFilteredByWriteAccessUsersActivityView extends
UsersActivityView {
  UserGroupListView allowedList;
  UserGroupListView notAllowedList;
  public SortedFilteredByWriteAccessUsersActivityView() {
    super();
  }

  public SortedFilteredByWriteAccessUsersActivityView(UsersManager
anUsersManager,ChatLocalApplication aChatLocalApplication){
```

```
    super(anUsersManager,aChatLocalApplication, false);
    this.createGUI();
  }

  public void createGUI(){

    allowedList=new UserGroupListView("Allowed",
    new utilities.DyceFilteredListModel(
    new SortedListModel(
    new
DDefaultListModel((utilities.ObjectListenerList)(this.getUsersManager().g
et(this.getUsersManager().chatParticipantsSlotName))),
    new
chatblocks.model.chatUser.ChatUserComparator()),(utilities.ListFilter)
    (new utilities.ListFilter(){
        public boolean satisfies(Object o){
          final ChatUser cu=(ChatUser)o;
          return
               ((Boolean)
              (new DisplayTransaction(){
                  public void txBody(){
                    result=new
Boolean(getUsersManager().getFloorControlManager().isAllowedToWrite(cu));
                  }
               }).doIt().result()).booleanValue();                    }
      })
    ),this);

    notAllowedList=new UserGroupListView("Not allowed",
    new utilities.DyceFilteredListModel(
    new SortedListModel(
    new DDefaultListModel((utilities.ObjectListenerList)
      (getUsersManager().get(this.getUsersManager().chatParticipantsSlotN
      ame))),new chatblocks.model.chatUser.ChatUserComparator()),
      (utilities.ListFilter)

    (new utilities.ListFilter(){
        public boolean satisfies(Object o){
          final ChatUser cu=(ChatUser)o;
          return
               ((Boolean)
              (new DisplayTransaction(){
                  public void txBody(){
                    result=new
Boolean(!getUsersManager().getFloorControlManager().isAllowedToWrite(cu))
;
                  }
               }).doIt().result()).booleanValue();                    }
      })
    ),this);
    this.addUserGroup(allowedList);
    this.addUserGroup(notAllowedList);

  }
}
```

Para mas información acerca de las clases empleadas aquí, puede consultar el manual de referencia de clases (el package es utilities) y ver el capítulo 7 referente a extensiones al framework DyCE.

Envío de Mensajes

Para poder escribir los mensajes y enviarlos, el usuario debe emplear el bloque visual ShipmentView. Vea la sección ShipmentView para mas detalles acerca de cómo esta compuesto este bloque visual de Chatblocks.

La clase ShipmentView es abstracta. Un componente Chatblocks concreto deberá subclasificar esta clase y definir un único método abstracto para utilizarlo:

```
public  void sendButton_action(ActionEvent e)
```

Un ShipmentView recibe este mensaje cuando el usuario presiona el control (por defecto un botón con el texto "send") destinado a enviar el mensaje escrito al MessagePool. Este método es empleado para construir el objeto mensaje del modelo, el cual deberá ser adaptado al dominio específico de la aplicación , y finalmente enviar el mensaje al ShimentManager (que es el modelo de este bloque visual).

Por ejemplo, un código para un ShipmentView simple puede ser:

```
public class SimpleShipmentView extends ShipmentView {
```

Suponga que tiene como modelo el SimpleShipmentManager, y basta con utiliza el bloque por defecto de la clase AttributeFortextFieldView:

El constructor se verá como:

```
 public SimpleShipmentView(SimpleShipmentManager
aNoReferenceShipmentManager, ChatLocalApplication
aChatLocalApp,AttributesForTextFieldView attributesView) {
    super(aChatLocalApp);
    this.setShipmentManager(aNoReferenceShipmentManager);
    }
```

Ahora, debe definir el método abstracto que enviará el mensaje al Shipment Manager

```
  public void sendButton_action(ActionEvent e) {
      String textToSend = this.textToSend.getText();
      this.clearTextToSend();//blanks the text field
         final String tts=textToSend;
```

Es necesario acceder al usuario actual y al Shipment manager, ambos son objetos compartidos por lo tanto, es obligatorio emplear transacciones:

```
final ChatUser cu= (ChatUser) ((new Transaction()
           {
           public void txBody()
```

```
            {
      result = getChatLocalApp().getCurrentUser();
            }
            }).doIt().result());


  final ChatMessage cm= (ChatMessage) ((new Transaction()
            {
            public void txBody()
            {
result=
getShipmentManager().createMessage(tts,cu,
 getTextAttributes(),
 getCurrentMessageType());
            }
            }).doIt().result());
```

Finalmente, el mensaje es enviado al Shipment Manager:

```
(new Transaction()
            {
            public void txBody()
            {
                  getShipmentManager().sendMessage(cm);
            }
            }).doIt();
```

Recepción de Mensajes

Chatblocks cuenta con un bloque visual concreto para procesar y mostrar los mensajes permitidos por el Reception Manager: el ReceptionView.

Si se desea emplear este bloque tal como esta en la aplicación, basta con el siguiente código en la clase ChatLocalApplication:

```
public  ReceptionView createReceptionView(){
    ReceptionView rv;
    ListMessageViewer mv = new ListMessageViewer(rv=new ReceptionView(
this.getReceptionManager(),this ),"List");

    rv.addMessageViewer(mv);

    mv.add(this.getPopupMenu());
    mv.addMouseListener(this.getMouseListener());
    mv.setListCellRenderer(new PaneMessageListCellRenderer());
    mv.setSelectionModel(new ToggleListSelectionModel());
   return rv;
            }
```

El ReceptionView es uno de los bloques más versátiles del framework, el programador puede configurar al reception view con cualquier objeto que implemente la interfaz MessageViewer. Cada MessageViewer ofrecerá una vista diferente de los mensajes a

mostrar, el ReceptionView es capaz de trabajar con varios MessageViewer a la vez, siendo el usuario quien puede elegir en tiempo de ejecución que vista desea emplear. De esta forma, como estos bloques son componentes locales. Cada usuario puede en su máquina emplear una vista diferente para los mismos mensajes que todos están visualizando.

El método empleado para agregar un objeto MessageViewer al ReceptionView es:

```
public void addMessageViewer (MessageViewer viewer)
```

Es posible agregar menús contextuales a los MessageViewer. Por ejemplo, si observa la clase L3Chat (correspondiente a una familia de aplicaciones construidas con Chatblocks), se puede ver como factoriza el comportamiento referente a estos menús con el código:

```
mv.add(this.getPopupMenu());
mv.addMouseListener(this.getMouseListener());
```

Los MessageViewer emplean un "renderer" (nombre que emplea Java para designar al objeto empleado para obtener una representación visual de otro objeto del modelo). En el caso de la clase ListMessageViewer, que ofrece una visualización en forma de lista de los mensajes enviados, el renderer puede ser fijado mediante el método:

```
mv.setListCellRenderer(new PaneMessageListCellRenderer());
```

Vea la referencia de clases de Chatblocks (el documento JavaDoc) para mas detalles de lo que debe implementar un objeto para satisfacer la interfaz MessageListCellRenderer.

También es posible elegir entre diferentes tipos de selección sobre los elementos de la lista, basta con crear el objeto apropiado que implemente la interfaz java ListSelectionModel. En este caso contamos con un objeto que permite seleccionar y deseleccionar algo de la lista de forma alterna:

```
mv.setSelectionModel(new ToggleListSelectionModel());
```

El manual de referencia de clases contiene todas las interfaces y métodos que se necesitan conocer cuando se quiere programar un ReceptionView.

Interfaz del Protocolo de Comunicación

El pane FloorControlView permite obtener información relacionada con el estado de la comunicación, y proporciona los controles necesarios para manejar el flujo de la conversación.

En caso de componentes Chatblocks simples, como el SimpleChat, este objeto puede llegar a tener una representación no visual debido a que el protocolo de comunicación empleado no requiere de controles o indicadores visuales para ejecutarse. En estas situaciones, el framework brinda un pane vacío denominado NoProtocolFloorControlView.

Cuanto más complejo sea el tipo de conversación virtual que se intente modelar, más complejo será el floor control manager del sistema, y por lo tanto más complejo se volverá su contraparte visual, es decir, el FloorControlView. Por ejemplo: se podría tener un protocolo en el cual el usuario deba hacerle saber a otro usuario que desea enviar un mensaje antes de tener la posibilidad de enviar el mismo.
En este caso, el FloorControlView deberá incluir un control cuya acción al ejecutarlo (digamos, presionar un botón), sea enviar una petición al usuario encargado de conceder los turnos.

Todas las decisiones acerca de cuando un control debe estar habilitado, y los cambios que reflejen el estado actual del protocolo deben ser delegadas al modelo de este bloque visual, es decir una subclase de FloorControlManager.

Misceláneas de la Interfaz de Usuario

Para mejorar y aumentar las capacidades que ofrece un componente Chatblocks, existen otros sub-bloques visuales que pueden ser personalizados o especializados. A continuación se detallan las clases involucradas y una breve descripción de su propósito:

Tabla 6.6. Objetos de la GUI

Dialog Boxes	Description
TypeSupportingView	Usado por el ShipmentView para el soporte de tipos
MessageViewer	Usado por el ReceptionView para visualizar los mensajes enviados
AttributeForTextFieldView	Usado para definir los atributos del texto de un mensaje

Tabla 6.7. Cuadros de Diálogo

Dialog Box	Description
ExportView	Diálogo empleado para exportar los mensajes
MessageEditDialog	Utilizado por el MessageViewer para editar un mensaje ya enviado
StatistiesView	Muestra información estadística de un usuario. Es el componente por defecto utilizado por el UsersActivityView

Tabla 6.8. Renderers

Renderer	Description
MessageTypeCellRenderer	Renderiza un MessageType, empleado por el TypeSupportingView
UserListCellRenderer	Renderiza un objeto ChatUser, empleado por la lista en el UserActivityView
DefaultListCellRenderer	Renderiza un objeto ChatMessage en una lista, usado por el ListMessageViewer
DefaultTreeCellRenderer	Renderiza un objeto ChatMessage en una lista, usado por el TreeMessageViewer

Capítulo 7. Desarrollo de Extensiones a DyCE

Uno de los objetivos principales de esta tesis consistió en la implementación del framework desarrollado en el capítulo anterior.
La implementación final elaborada en java y basada en DyCE permite satisfacer los requerimientos tanto del usuario como del desarrollador (Véase capítulo 2).
En el proceso de diseño e implementación llevado a cabo, fueron surgiendo otro tipo de necesidades que debieron ser satisfechas para poder continuar con el desarrollo del framework.
Estas nuevas necesidades se presentaron debido al hecho de que estamos trabajando con un producto experimental que aún hoy se encuentra en evolución como lo es el framework DyCE.
Uno de los aspectos más "frágiles" que presenta DyCE es que no posee componentes visuales especializados para ser empleados de la misma forma que cualquier otro componente estándar de Java (los componentes swing no poseen características de aplicaciones groupware). Por lo tanto al intentar desarrollar un framework que le brinde al usuario final la posibilidad de simplemente unir bloques para tener un componente groupware funcionando, nos topamos con la dificultad de que los bloques debían ser tan flexibles como para poder ser extendidos sin problemas, pero a la vez lo suficientemente cerrados, como para que el programador los pueda emplear de la misma forma que opera con un componente Swing normal.

Al trabajar con un paradigma groupware, donde el modelo compartido de la aplicación obliga a imponer nuevas reglas para su manipulación, optamos por la creación de un mecanismo intermedio que opera entre el modelo compartido y la interfaz visual de un componente Swing tradicional. Estos objetos intermedios funcionan como wrappers de los objetos compartidos y permiten emplear los controles Swing normales prácticamente sin necesidad de tener que preocuparse de que debajo se esta empleando un modelo compartido.

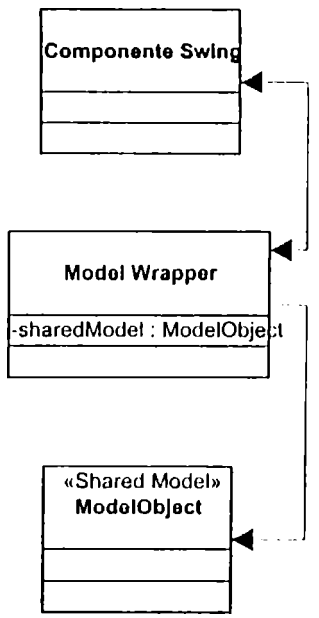


Figura 7.1 Relación entre componente visual, modelo y objeto compartido

Los componentes visuales de modelos complejos como la Jlist y el Jtree son los mas empleados. Sobre los modelos de estos dos objetos, la interfaz ListModel y TreeModel, y los renderers asociados creamos los objetos.

7.1 La Clase ObjectListenerList

Esta clase proporciona las siguientes funcionalidades:

1. Es capaz de contener ModelObject de la misma forma que se puede operar con un Vector Java normal. DyCE presenta ciertos inconvenientes cuando se anidan objetos compartidos, por lo que se recomienda usar los ObjectID de los objetos en lugar de los objetos en sí, si el programador emplea esta clase en lugar del Vector provisto por DyCE, se evita tener que trabajar con estos ObjectIDs.
2. Como función principal de un objeto de esta clase, se proporciona un mecanismo de notificación de eventos como pueden ser agregar o quitar un elemento. Pero a diferencia de los Vectors de Java o incluso de DyCE, con este objeto es posible recibir notificaciones de modificaciones sobre los slots de los objetos contenidos en el ObjectListenerList.

7.2 La clase DDefaultListModel

Si bien, ObjectListenerList brinda la funcionalidad básica de la clase Vector, mas un mecanismo mas avanzado de notificación de eventos. No es posible emplear directamente un objeto así como modelo de un JList, o cualquier otro componente Swing de Java que emplee listas. Es necesario otro objeto que se comporte como un ListModel y que tome

como modelo un ObjectListenerList, es este el caso de la clase DDefaultListModel, que implementa la interfaz java ListModel.
Empleando estos objetos, el programador se desliga del hecho de tener que preocuparse por los accesos a objetos compartidos y por los listeners asociados, todo esto se realiza de forma automática.

7.3 La clase ListToTreeModelAdapter

En ciertas ocasiones es deseable mostrar ciertos datos, por ejemplo, el contenido de una lista, en una representación con estructura de árbol. En tales caso se puede emplear un objeto de la clase ListToTreeModelAdapter, el cual se comporta como un DDefaultListModel, pero adaptando el ObjectListenerList a la interfaz Java TreeModel (que es empleada por el componente Swing JTree para mostrar modelos de estructura jerárquica).
Para utilizar un ListToTreeModelAdapter, el programador debe especificar la lista y un TreeCreator, un objeto que es capaz de construir un nodo del árbol, a partir de un elemento de la lista.

Capítulo 8. Conclusiones

8.1 Contribuciones al Estado del Arte

Teniendo en cuenta el mercado actual, se pueden notar fácilmente las deficiencias que presentan las aplicaciones existentes para aquellos ambientes en donde se desea emplear las herramientas de comunicación virtuales para la enseñanza o la discusión de ideas entre colegas:
La rigidez de dichos sistemas, y la falta de una herramienta en donde sea el usuario quien decida como quiere llevar a cabo la comunicación, son los principales problemas.

Con la construcción de este framework, se da un primer paso para facilitar un ambiente de experimentación más adaptable y configurado de acuerdo al propósito de cada tipo de discusión por los medios virtuales que se pretenda encarar.
Si bien, en esta implementación la flexibilidad máxima solo se consigue al nivel del programador (no es un framework black box, y por lo tanto la modelización no puede llevarse a cabo de forma totalmente automática mediante los conocidos asistentes o agentes inteligentes), ofrece una alternativa diferente, de bajo costo y que deja abierta la puerta para profundizar aún mas en el área de la comunicación humana a través de medios virtuales.

Actualmente, Chatblocks esta siendo empleado por un conjunto de sociólogos y psicólogos, que estudian diferentes protocolos y mecanismos principalmente para lograr la mejor forma de aprendizaje a distancia. El Fraunhofer, es uno de los institutos interesados en el aprovechamiento de este framework para propósitos educativos. Sobre todo en la posibilidad de extenderlo con otros medios audiovisuales, como puede ser una pizarra colaborativa o un presentador de exposiciones (similar a un PowerPoint colaborativo)

8.2 Trabajos Futuros

Chatblocks es el puntapié inicial para abrir el camino a otras ramas de estudio relacionadas con los protocolos de comunicación y su automatización.
Podemos clasificar los trabajos en dos grupos: Generación Automática de Aplicaciones y Especificación Formal de Protocolos de Comunicación. A continuación se describen ambos proyectos y se presenta una tercera alternativa de desarrollo relacionada con la generación de aplicaciones.

8.2.1 Generación Automática de Chats (Entorno de Desarrollo, IDE)

Chatblocks es un framework de caja de blanca, es decir, requiere del conocimiento de su estructura interna, y la extensión y desarrollo de aplicaciones se realiza mediante la sub-clasificación de elementos existentes que sirven de base. Sin embrago, teniendo en cuenta los factores configurables y extensibles que brinda el framework, es posible obtener una versión cerrada del mismo, es decir, algo del tipo black-box, de forma de poder un crear entorno de desarrollo de aplicaciones que automatice la generación de las mismas. Mas aún, el entorno puede ser de carácter visual, al igual que los provistos para el desarrollo de aplicaciones java empleando componentes Swing (por ejemplo el IDE de JBuilder).
Estos ambientes podrían basarse en los componentes visuales del framework Chatblocks, para brindarle al programador un conjunto de propiedades configurables.
Para lograr esto, un primer paso seria adaptar los bloques gráficos a la especificación de los componentes JavaBeans. Luego solo restaría la construcción de un editor mas sofisticado que los empleados para estos Beans, de forma de poder abarcar los aspectos más fundamentales de la aplicación como son el carácter de modelo compartido y especialmente la definición o implementación del protocolo de comunicación del sistema.

La existencia de un IDE para la generación de componentes chats, puede significar un gran avance para aquellos investigadores que desean experimentar de forma rápida y sencilla con mecanismo alternativo para llevar adelante una conversación. Con una herramienta así, pueden fácilmente construir sus aplicaciones, ejecutarlas, sacar conclusiones, y luego simplemente ajustar algunos parámetros, para tener listo un nuevo sistema con alguna variación, digamos en el protocolo, y luego tras realizar nuevas pruebas, poder comparar resultados.

Volver a Chatblocks un framework black-box, no debería imponer límites al programador para que pueda realizar extensiones de la misma forma que si este siguiera siendo white-box. De lo contrario, se perdería la gran flexibilidad y características dinámicas con las que cuenta hasta ahora.

El aspecto más difícil de diseñar e implementar es la construcción de un editor práctico para los protocolos. La principal dificultad radica en que no existen los formalismos necesarios para la definición de dichos protocolos, por lo tanto traducir una especificación de protocolo a un lenguaje visual se vuelve prácticamente imposible debido a la gran cantidad de ambigüedades y cuestiones sin resolver que el lenguaje natural provoca.

Debido a estos inconvenientes parece una consecuencia lógica que se intente construir la mencionada notación formal para los protocolos, la cual es sintéticamente presentada a continuación

8.2.2 Notación Formal para los Protocolos de Comunicación

El objetivo es conseguir una notación que permita especificar el protocolo a implementar en un determinado chat.

Esta notación debe ser lo bastante amplia como para permitir especificar cualquier tipo de chat pero a la vez lo suficientemente sencilla como para que pueda ser utilizada por personas ajenas o muy poco relacionadas al mundo de la informática. Esta notación tiene que estar libre de ambigüedades.

Factores involucrados

En primer lugar, es necesario aclarar que factores dentro de un chat son los que se desean representar.

Dentro de estos se encuentran los participantes de la conversación, los cuales poseen características que determinan cual va a ser su posterior desempeño. Estas características se agrupan y definen Roles, los cuales, representan a un grupo de personas que poseen características comunes. Cada participante del chat, debe pertenecer indefectiblemente a un rol.

```
El desempeño esta regido de acuerdo a las acciones en las que estén
involucrados los roles que participan en el chat.
```

La aplicación no reconoce funcionalidad al nivel de usuario, sino que esta factorizada al nivel de Roles de usuario. La funcionalidad permitida por la aplicación esta dada por las tuplas (Función, Estado de la Aplicación, Rol habilitado).

Las acciones estarán basadas en las posibles contribuciones y en los momentos en los que puedan hacerse. Las mismas desencadenarán el flujo de interacción entre los roles

Teniendo en cuenta estos factores, surgió como una primera posibilidad la representación del protocolo basada en la idea de un autómata, pero sin entrar en notaciones específicas y conceptos asociados a la teoría en autómatas:

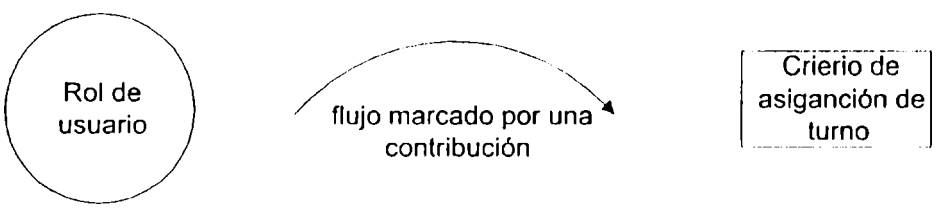


Figura 8.1 Ejemplo de una posible notación para graficar protocolos de comunicación

Se pueden emplear círculos que representen roles unidos por flechas, las cuales son llamadas contribuciones.

Una flecha que sale de un rol hacia otro significa que el primero rol puede efectuar una contribución del tipo que indica la flecha, y en caso de realizarla el próximo que podrá contribuir será el rol de destino. Esto determina el flujo natural del chat.

En definitiva, lo que se desea representar con este tipo de notación, es como funciona el control de flujo en una conversación de un chat en particular
Las uniones entre estos estados estarán constituidas por transiciones, las que luego, determinaran el flujo de diálogos para cada tipo de chat.

Con estas ideas se puede plantear un lenguaje formal para un protocolo básico de comunicación. Mediante nuevas representaciones gráficas se le puede ir agregando mas información o nivel de detalle como por ejemplo, la existencia de turnos (pedir y otorgar el derecho de hacer una contribución), características que permitan especificar relaciones entre las contribuciones (por ejemplo indicar que una contribución es respuesta de otra) y por último la posibilidad de asociar tipos (Mensajes con tipo)

8.2.3 Extracción y Modificación en Run-Time del Modelo o la Interfaz de una Aplicación

Otra idea mas ambiciosa es la construcción de un sistema que permita la extracción del modelo de un componente Chatblocks y que permita modificarlo en tiempo de ejecución, algo similar a tener un inspector de los bloques JavaBeans.

Por ejemplo, si se tiene un determinado componente Chatblocks, cuyo protocolo de comunicación posee ciertos parámetros ajustables, digamos, la automatización de la concesión de los turnos a los roles. Mediante un sistema como este se podría alterar ese parámetro en tiempo de ejecución, de modo que sin necesidad de reiniciar la aplicación ahora los turnos pasen a darse automáticamente (si suponemos que es ese el parámetro alterado).

Por supuesto, un sistema de estas características impondrá un número mas fuerte de restricciones sobre el componente Chatblocks. Por lo tanto no podrá ser aplicado sobre cualquier aplicación. El grado de ajuste en tiempo de ejecución debe hacerse explícito a nivel del programador.
Una posibilidad es definir propiedades públicas (similares a los fields public de los JavaBeans), por ejemplo con el prefijo *dinamic* que le indicarían al ambiente de desarrollo que su valor puede ser alterado en tiempo de ejecución. De esta forma el protocolo de comunicación debe ser implementado de acuerdo al valor de las propiedades dinamic, es decir, utilizar estos valores para construir una meta-maquina de estados que define protocolos diferentes para cada valor.

Esta área de investigación recién esta siendo analizada.

Apéndice A. Experiencias de Uso

La implementación de Chatblocks que se presenta en esta tesis ya esta siendo utilizada para la investigación y construcción de componentes DyCE que sirven como parte de un sistema cooperativo de aprendizaje a distancia denominado L^3 (Long Life Learning). El instituto Fraunhofer, solicitó al LIFIA la construcción de una familia de componentes prototípicos de sistemas chats, los cuales fueron desarrollados con Chatblocks y que se presentan en este trabajo como ejemplos de la utilización del framework.

A continuación se ofrece una breve descripción de en que consiste cada uno de los componentes implementados, haciendo hincapié en el protocolo que utilizan. Ya que la funcionalidad aparte de ello es similar en todas.

A.1 Simple Chat

Este es chat común empleado en capítulos anteriores como ejemplo de cómo construir un sistema de forma sencilla con Chatblocks. No posee un protocolo explícito de comunicación, los participantes no asumen ningún tipo de rol, y la visualización de los mensajes es en forma de lista ordenados por orden de aparición (envío).
La parte compartida, **SimpleChatApplicationModel** es subclase de **ChatApplicationModel**, utilizando los siguientes bloques:

- *ReceptionManager*
- *UsersManager*
- *NoRoleRoleManager*
- *NoProtocolFloorControlManager*
- *NoReferenceShipmentManager*

La parte local de la aplicación, **SimpleChat,** es una subclase de **ChatLocalApplication**, emplea los bloques:

- *UsersActivityView*

- *NoProtocolFloorControlView*
- *NoReferenceShipmentView*
- *ReceptionView*: donde el *viewer* es una instancia de **ListMessageViewer**

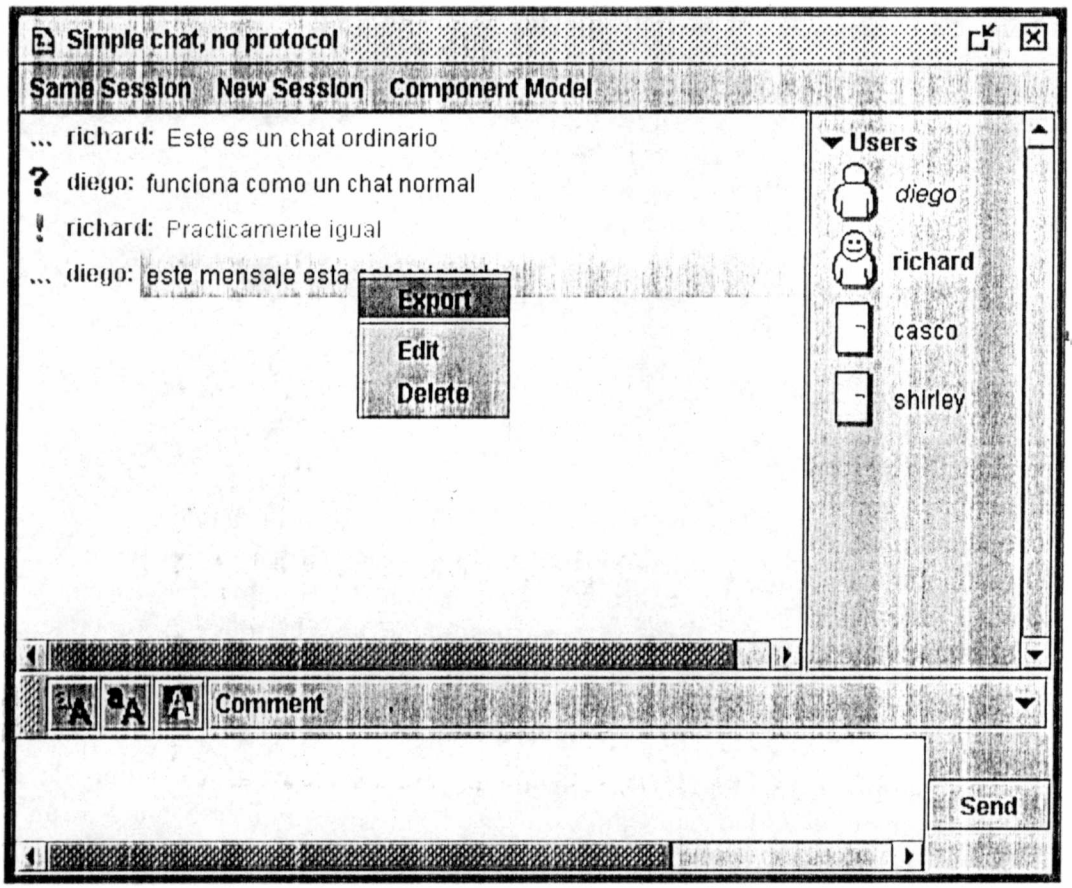


Figura A.1. Simple Chat

A.2 Pro-Contra

Este componente presenta un procotolo de comunicación que pretende emular un debate entre dos posturas opuestas. Por ejemplo, podría ser una discusión entre dos facetas políticas o religiosas.
En una discusión de este tipo, se pretende que ambas partes puedan exponer sus ideas de manera alternada. Es decir una vez que una de las partes presenta una idea o concepto, solo la otra parte puede responder con otro mensaje.

En términos de un sistema Chatblocks, participan tres roles: PRO, CONTRA y HEARER que solamente pueden comentar en cualquier momento, pero son personas que no están en ninguno de los dos bandos

La idea es que entre el PRO y el CONTRA exponen alternadamente las exposiciones, de manera que hable un PRO, después un CONTRA, después un PRO, y así siguiendo la secuencia. Los HEARER pueden hacer comentarios en cualquier momento, lo único que se tiene que respetar es el orden entre los dos privilegiados (PRO y CONTRA).

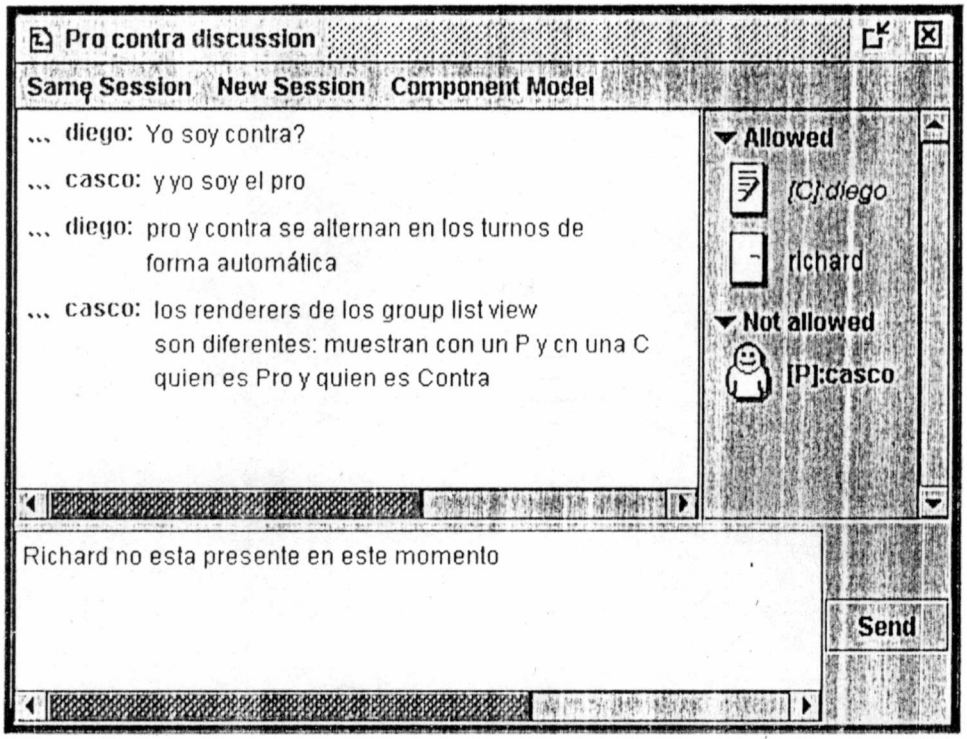


Figura A.2. Pro-Contra Chat

La clase principal involucrada en este sistema es ProContraFloorControlManager, que es la que define el protocolo descrito.

A.3 Human Mediated Discussion

En este sistema, se plantea una discusión entre dos o más participantes, que deberán solicitar permiso para poder pronunciar sus mensajes. Esto implica la existencia de un usuario cuya función en el sistema, es simplemente oficiar de mediador, otorgando los turno según crea conveniente.

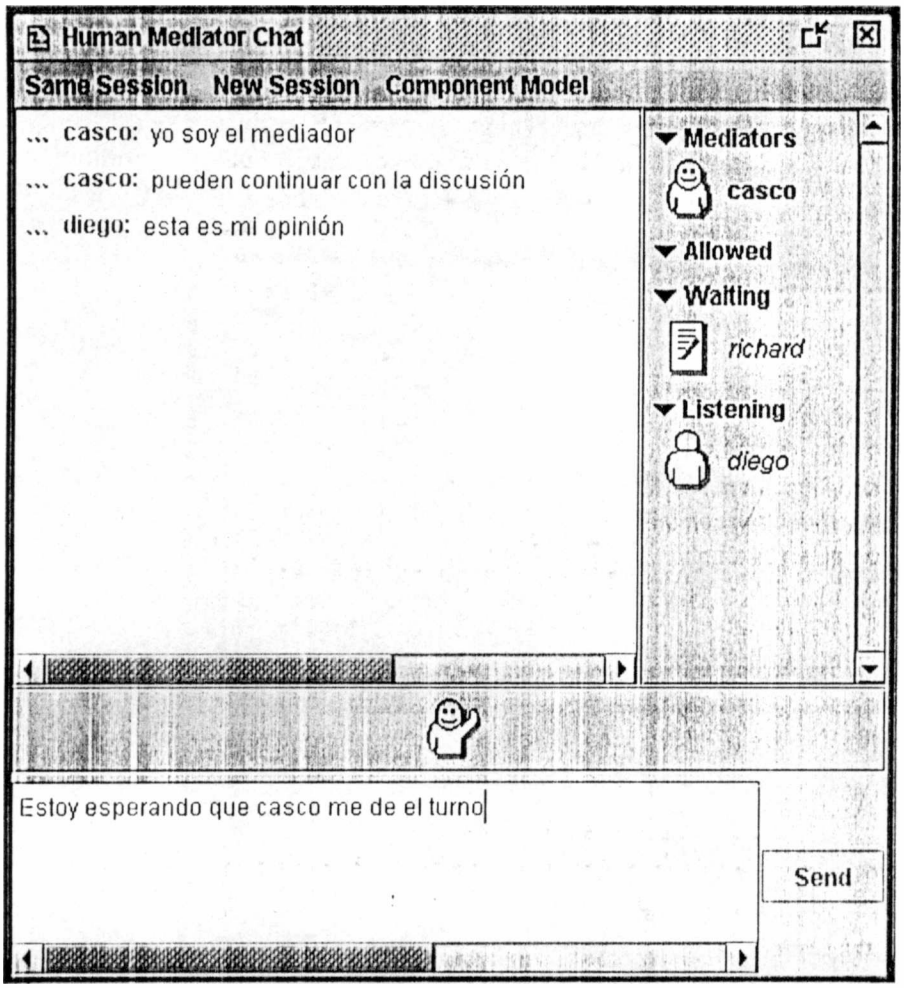


Figura A.3. Human mediated discussion

La implementación requiere definir 2 roles: Participant, Mediator. Una subclase especifica de FloorControlManager: *HumanMediatoFloorControlManager*. Y la contraparte visual del floor control, el *HumanMediatorFloorControlView*

A.4 Discussion with Auto-Tutor

En este chat, el protocolo de comunicación es similar al anterior, con la diferencia de que en este caso, no es necesaria la participación de un usuario como mediador. En su reemplazo, el protocolo es el encargado de determinar a quien le corresponde el derecho de enviar un mensaje.

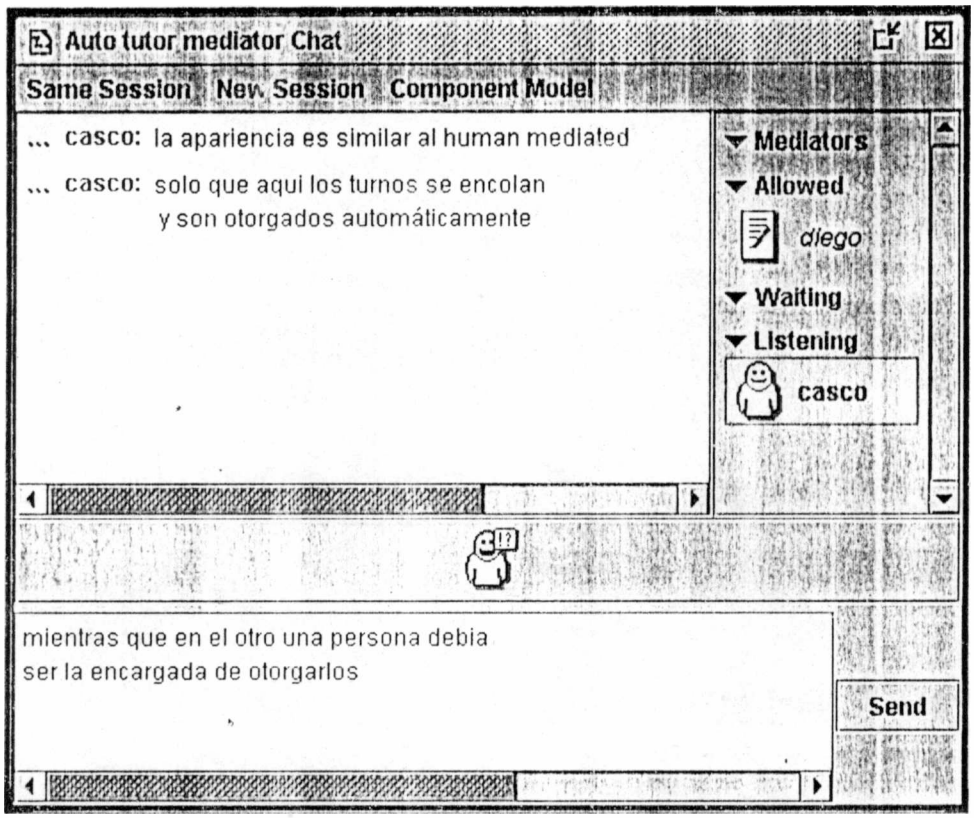


Figura A.4. Discusión con Auto-Tutor

La implementación de este protocolo, toma la lista de peticiones y otorga los turno en forma FIFO (una cola). Para los participantes la interfaz de usuario y la forma de operar es la misma, deben solicitar un turno y esperar hasta que sean habilitados para enviar un mensaje.

A.5 Threaded Chat

En este sistema, el protocolo de comunicación es indistinto, es decir, la implementación concreta emplea un NoProtocolFloorControlManager, pero éste puede ser reemplazado por cualquier otro, dependiendo de que tipo de discusión se lleve adelante.
La principal diferencia con el resto de los chats, es que emplea una clase especial de contribución, instancias de ThreadChatMessage, que admiten una referencia a un mensaje previamente enviado al pool de mensajes. Es decir, con este sistema es posible ir creando diferentes hilos de conversación, simplemente eligiendo cual es el padre de cada mensaje a enviar. Esta relación parent-child, es mostrada en el ReceptionView por medio de una estructura de árbol (aunque el usuario puede personalizar esto, y si quiere decidir ver en forma de lista los mensajes).

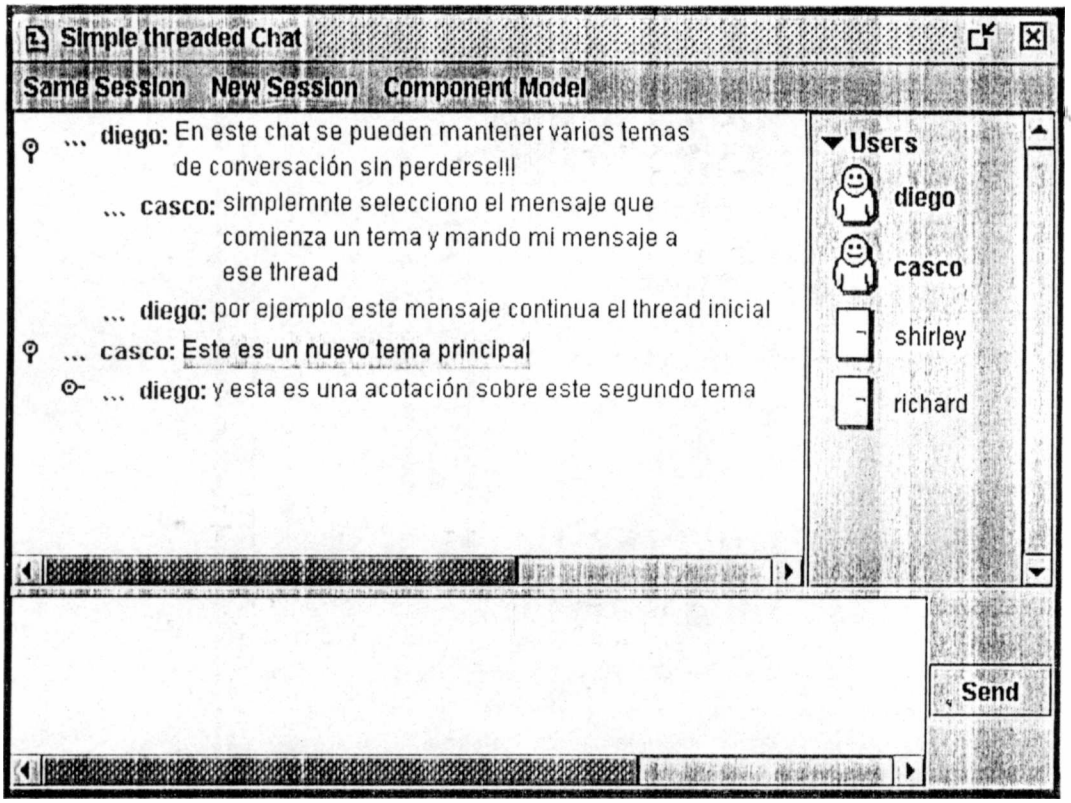


Figura A.5. Threaded Chat

Los bloques principales que deben tenerse en cuenta aquí son el TreeMessageViewer (como sub-bloque del ReceptionView básico), y el ShipmentView y ShipmentManager, ya que deben tener en cuenta la creación de la referencia al mensaje que el usuario haya seleccionado para crear y enviar el nuevo mensaje.

A.6 Learning Protocol Chat

Este chat trata de emular una cuarto de clases, donde existen los roles: Student, Teacher.
Además los mensajes poseen tipos: Comment, Explanation o Question.
La discusión se lleva a cabo de la siguiente forma:
El turno para enviar mensajes es pasado en forma de cola circular entre los estudiantes presentes.
Un estudiante puse pasar el turno al siguiente estudiante sin formular un mensaje, o puede enviar al resto una nueva contribución del tipo Question o Comment.
Si el mensaje enviado es del tipo Question, entonces el sistema cede el turno al Teacher para que éste emita un mensaje del tipo Explanation.
En cambio, si el alumno pasa, o envía un mensaje con el tipo Comment, entonces el control simplemente pasa al siguiente estudiante.

De esta forma, se va invitando a todos los estudiantes a participar de la discusión y a que expresen sus inquietudes para que el profesor a cargo las responda.

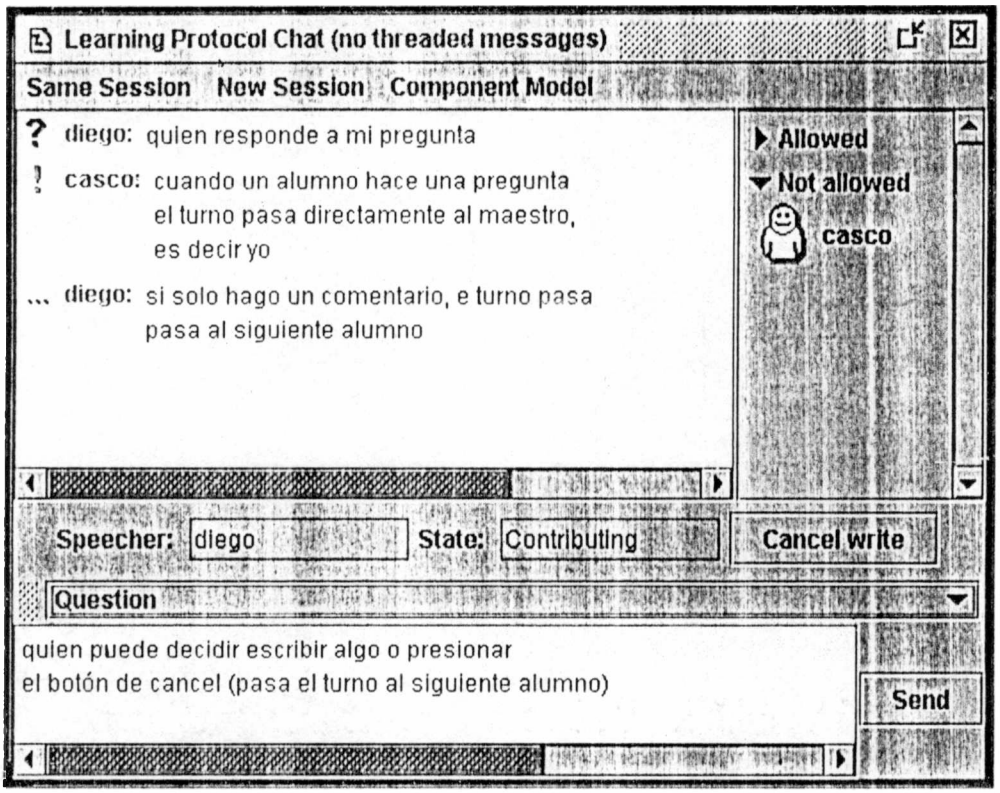


Figura A.6. Learning Protocol Chat

Los bloques principales en este sistema son el FlooControlManager y el TypeSupportingView (incluido en el ShipmentManager)

A.7 Snap Chat

El Snap chat es la herramienta más compleja creada con Chatblocks. Consiste en realidad de dos sistemas que interactúan entre sí: una aplicación chat tan compleja como cualquiera de las ya expuestas, y una pizarra colaborativa donde los participantes pueden mostrar y editar documentos.

La principal ventaja de tener asociada una herramienta de chat a la pizarra, radica en que se puede entablar una discusión (con el protocolo que se desee) acerca del contenido de algún documento mostrado en la pizarra.

La discusión se realiza creando *referencias* sobre alguna parte del documento. Es decir, cuando un usuario desea realizar algún comentario de alguna imagen o párrafo mostrado en el documento, simplemente lo selecciona y activa el comando **marcar** con el que aparecerá el bloque ShipmentView donde podrá escribir y enviar un mensaje relacionado a la sección del documento marcada. El resto de los usuarios verá un nuevo mensaje en el ReceptionView, y bastará con seleccionarlo de la lista, para que puedan ver mediante un recuadro y una flecha a que zona del documento esta haciendo referencia.

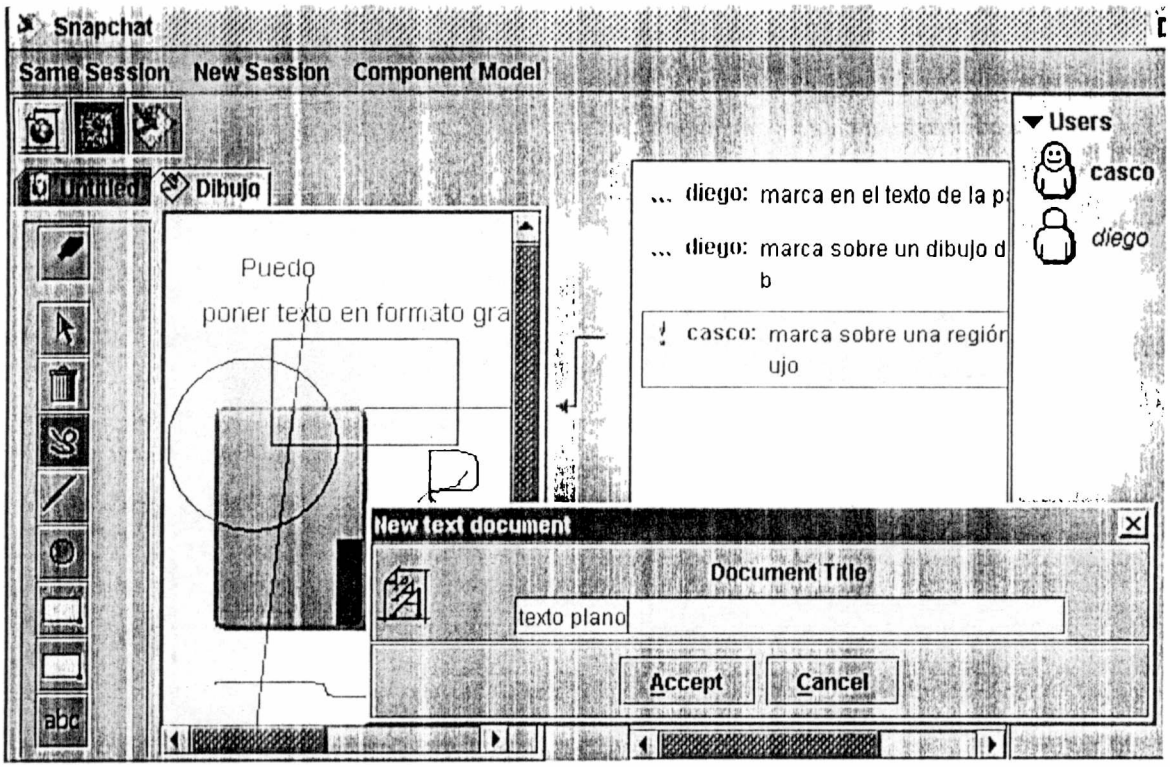


Figura A.7. Snapchat. Marcas sobre un documento de dibujo

En esta aplicación los bloques Chatblocks principales son: el ReceptionView, que se le agregó la pizarra colaborativa; el ShipmentView (y su manager) y casi la totalidad del modelo compartido.

Snapchat consta de otras funcionalidades que van mas allá de lo que concierne a Chatblocks (por ejemplo la forma en que se muestran las marcas, los tipos de documentos, etc.)

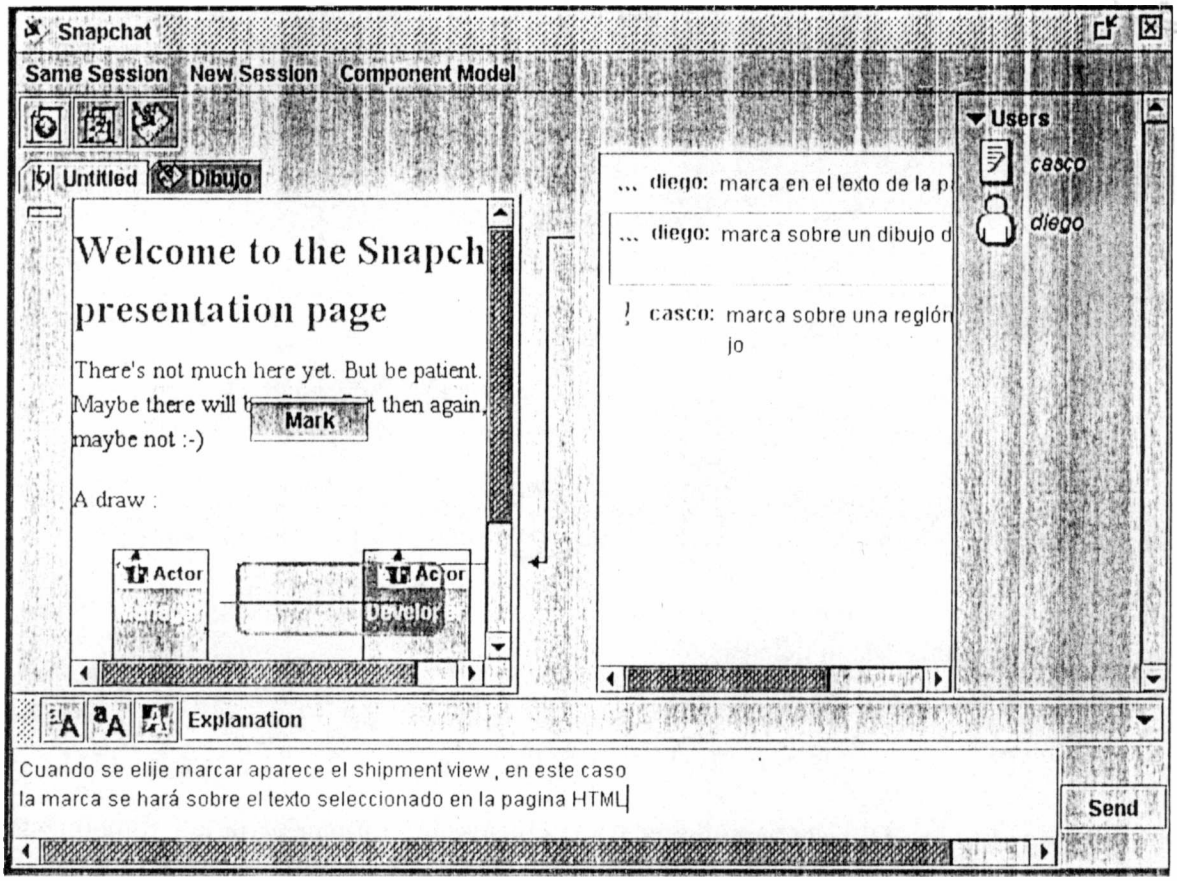


Figura A.8. Snapchat. Marcas sobre documento HTML

Apéndice B. Diagramas del Diseño del Framework

En este apéndice podrá encontrar los diagramas del diseño general del framework.

El modelo de datos lo componen los objetos que preservan la información generada por los usuarios durante una sesión del sistema.

El modelo compartido esta formado por aquellos objetos que son persistentes en la base de datos de DyCE, es decir, que serán replicados en cada cliente.

Los bloques locales del sistema componen los objetos que forman parte del componente groupware que sirve de interfaz al usuario para acceder a los objetos compartidos.

En la sección **"Construcción de Componentes Chatblocks"** pueden verse los diagramas que corresponden a los sistemas desarrollados con Chatblocks

Por último, se ofrecen unos bosquejos con una notación similar a la que se describe en la sección de trabajos futuros referente a notación formal de protocolos, presentando los gráficos correspondientes a cada uno de los protocolos implementados en los componentes implementados.

B.1 Modelo de Datos

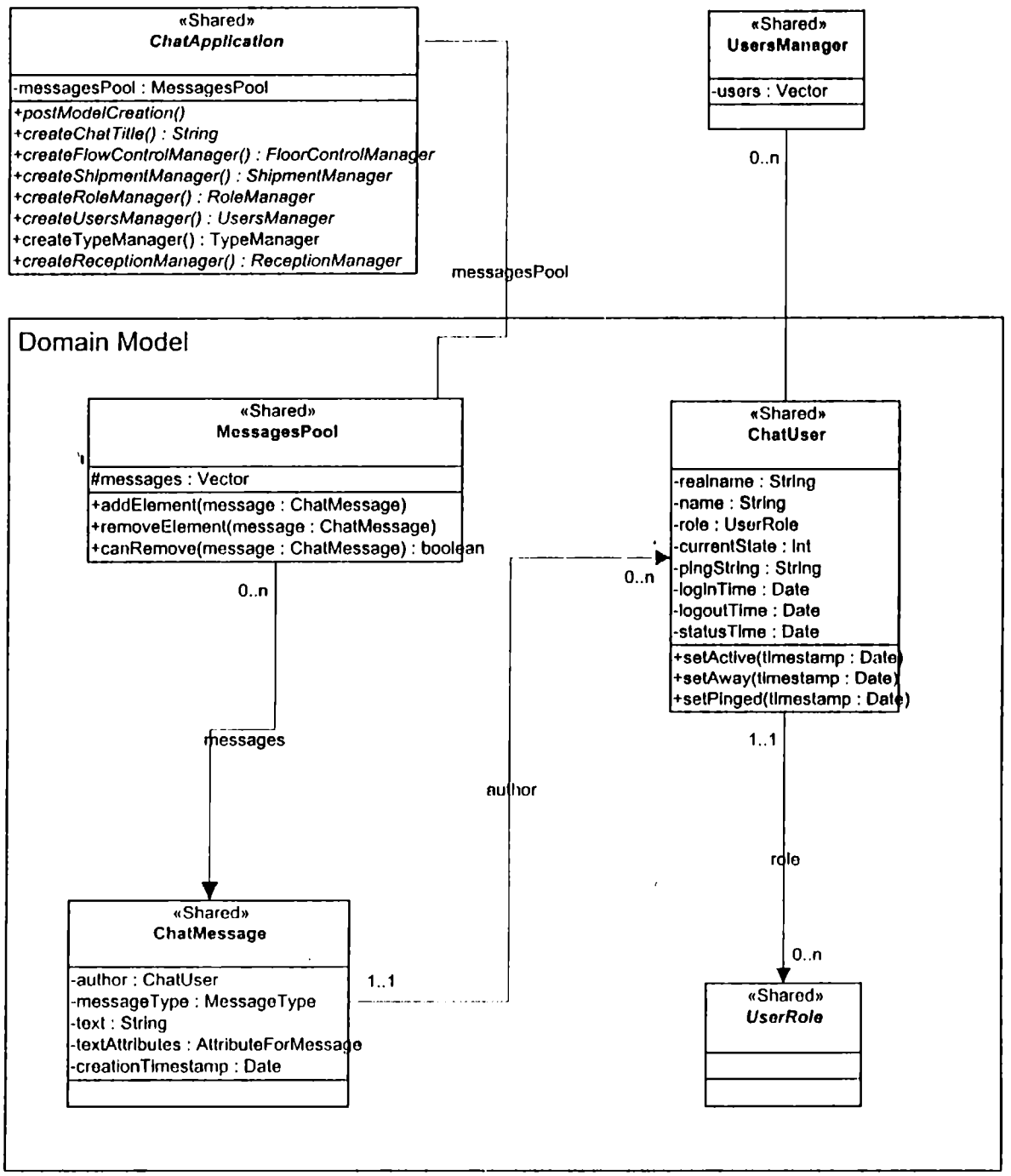


Diagrama B.1 Modelo de Datos

El modelo del sistema esta contenido en el objeto instancia de ChatApplicaction, el modelo de datos esta compuesto de la lista de mensajes enviados contenidos en el MessagesPool, y de la lista de usuarios del sistema administrada por el UsersManager.

B.2 Modelo Compartido

El siguiente diagrama muestra los bloques que componen el modelo compartido del sistema:

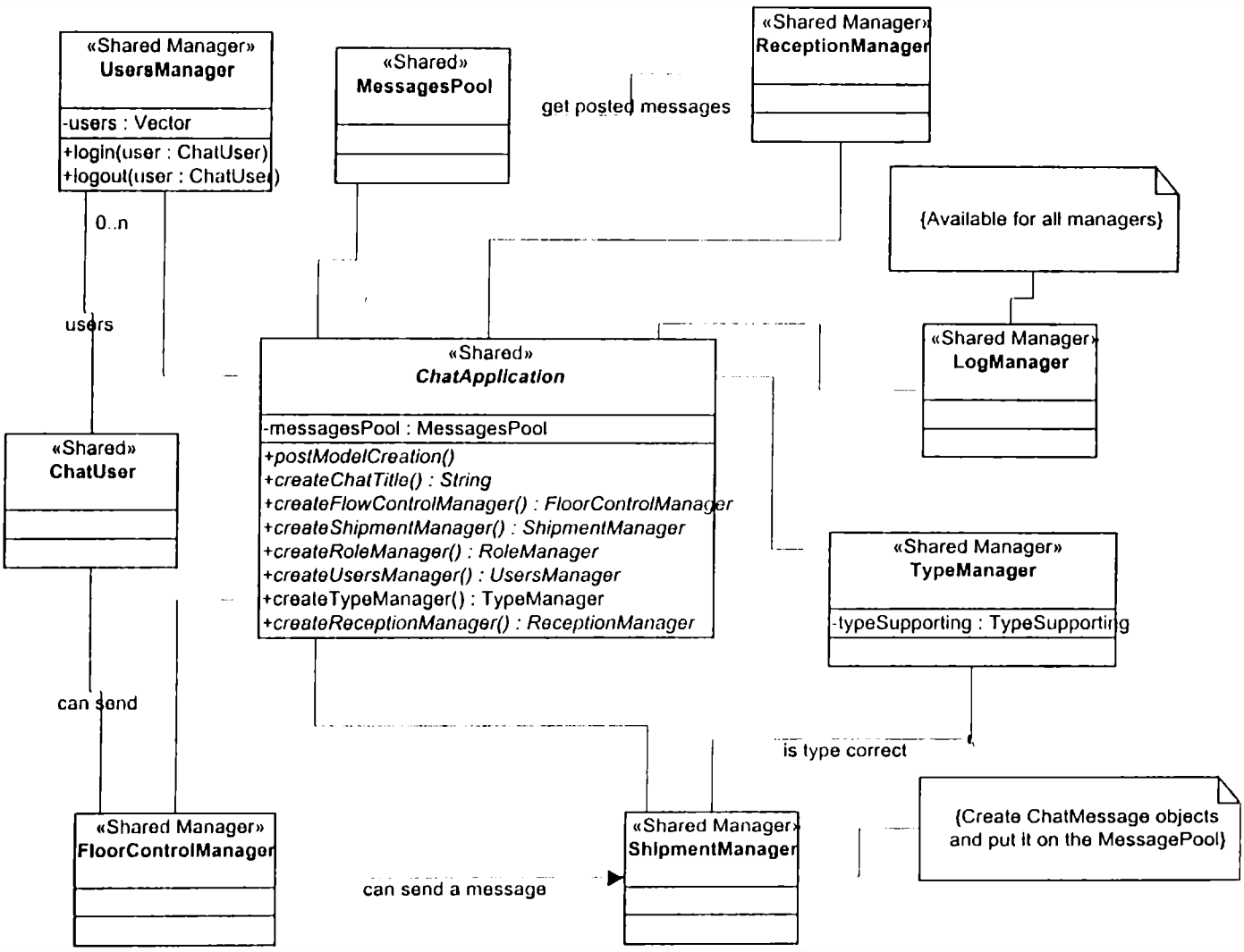


Diagrama B.2 Modelo Compartido: Managers del Sistema

Se han marcado como "Shared Managers", a los bloques Chatblocks que realizan la lógica compartida del sistema, manipulando los objetos del modelo de datos.
El LogManager esta disponible para todos los otros managers, de modo que desde cualquier parte del sistema se pueden registrar acciones que se crean convenientes. En particular el UsersManager registra mediante el LogManager los eventos de entrada y salida de usuarios del sistema (login logout).

B.3 Bloques Locales del Sistema (View Managers)

Los bloques locales son aquellos que sirven de interfaz al usuario del componente para interactuar con el modelo compartido de managers y de datos. Cada manager del modelo

compartido se encuentra asociado a una vista que contiene la representación visual del bloque y la lógica local del sistema

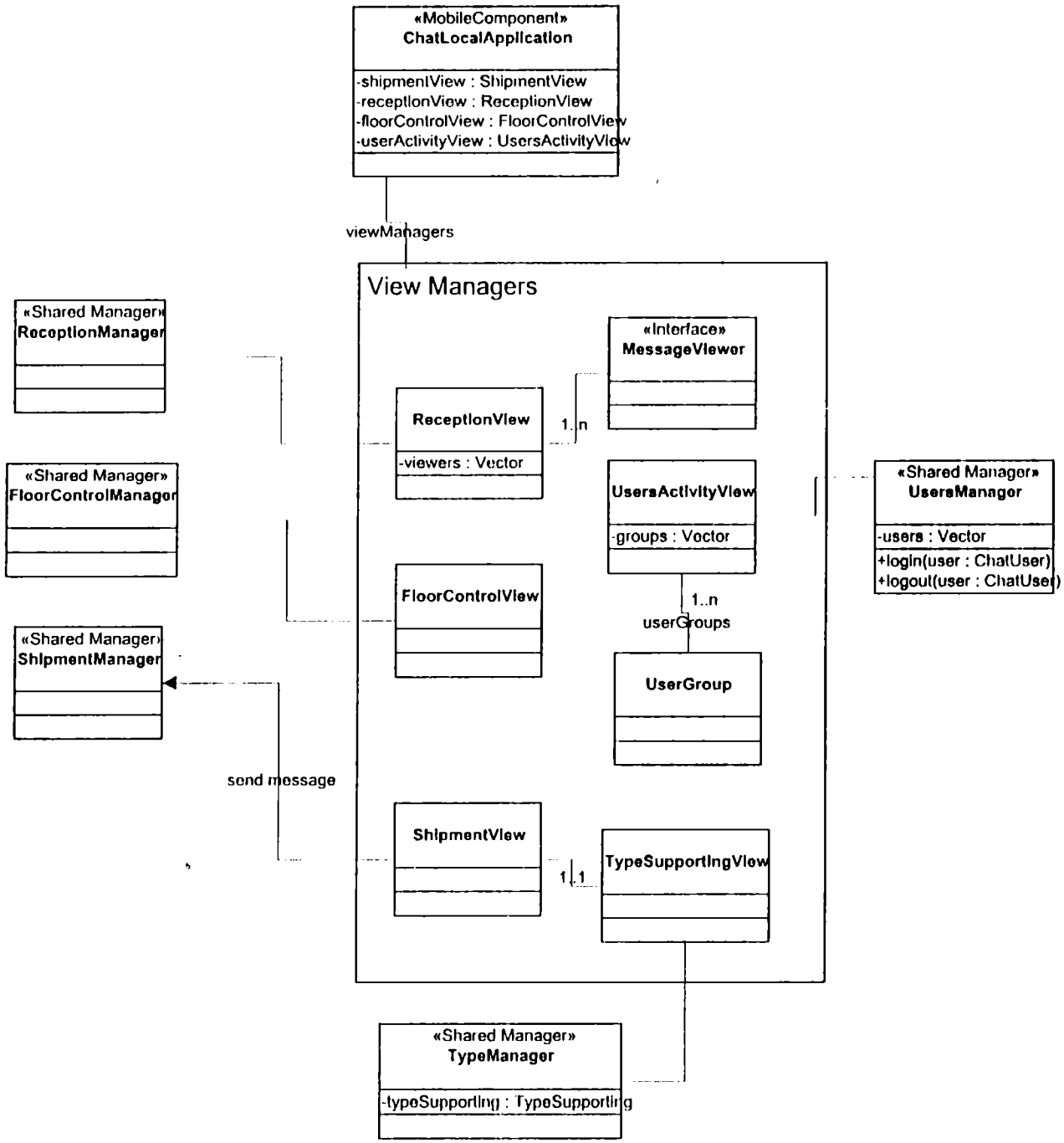


Diagrama B.3 Bloques Locales del Sistema (View Manager)

B.4 Diagramas del Diseño

UsersManager

El UsersManager contiene la lista de usuarios disponibles para comenzar una sesión. Diferencia entre usuarios sin privilegios, y usuarios administradores. Registra las entradas y salidas de usuarios del sistema.

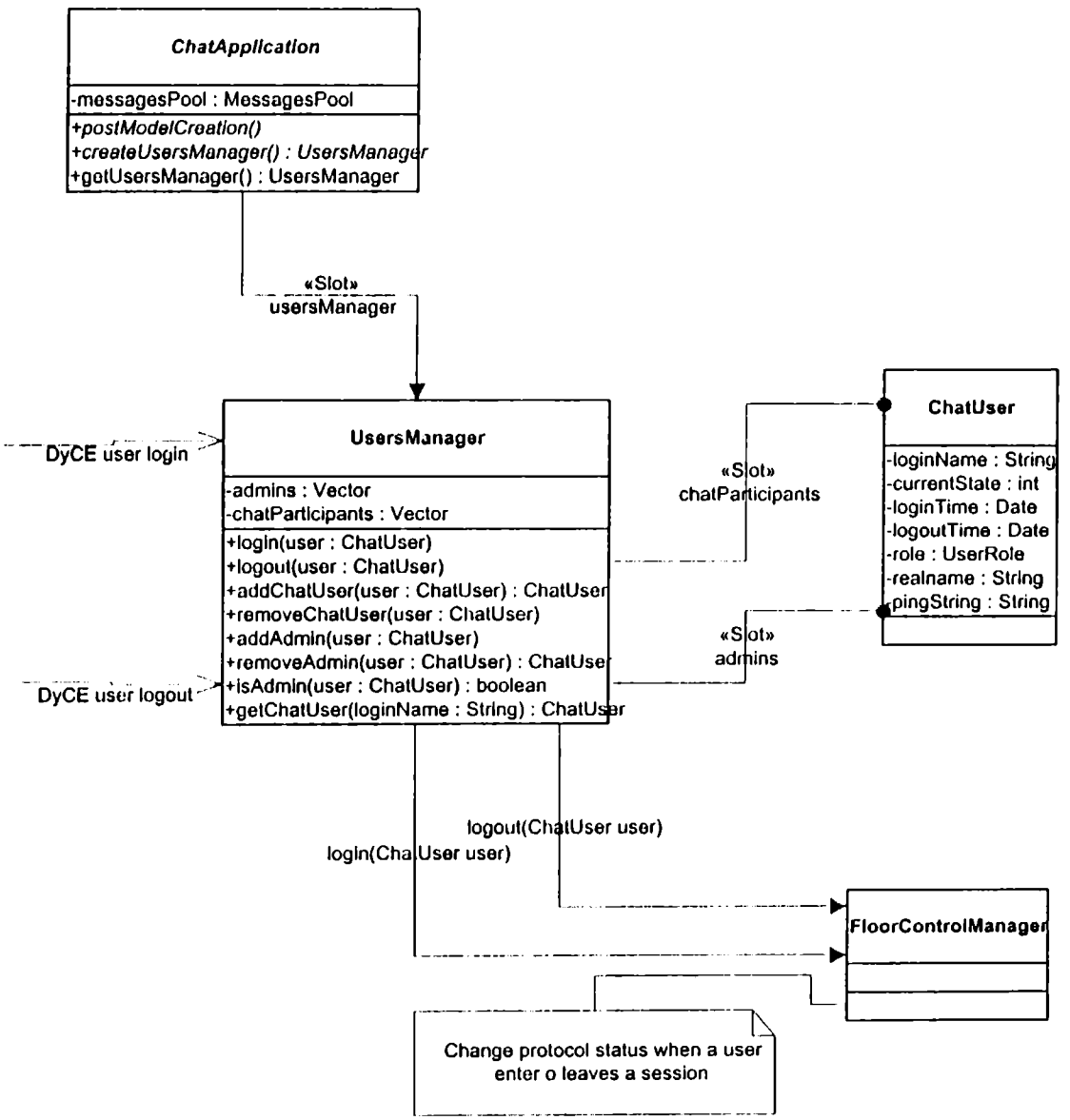


Diagrama B.4 UsersManager

Los usuarios Chatblocks son identificados unívocamente con los usuarios DyCE mediante el login de cada uno. Cuando un usuario nueva entra al sistema, el UsersManager chequea que dicho usuario este incluido como administrador o como participante, es decir, que este

presente en una de las dos lista de usuarios que maneja el UsersManager. Además se encarga de avisar al FloorControlManager que el usuario esta online, para que éste tome las medidas apropiadas. Cuando un usuario DyCE abandona la sesión, el respectivo usuario Chatblocks cambia al estado offline, pero no es removido de la lista de participantes.

ShipmentManager

El ShipmentManager se encarga de crear y enviar nuevos objetos ChatMessage al MessagePool. Para ello requiere interactuar con el FloorControlManager, para saber si el autor del mensaje tiene permiso para enviar el mensaje. Además debe comprobar la correctitud del tipo asignado al nuevo mensaje, para lo que necesita interactuar con el TypeManager.

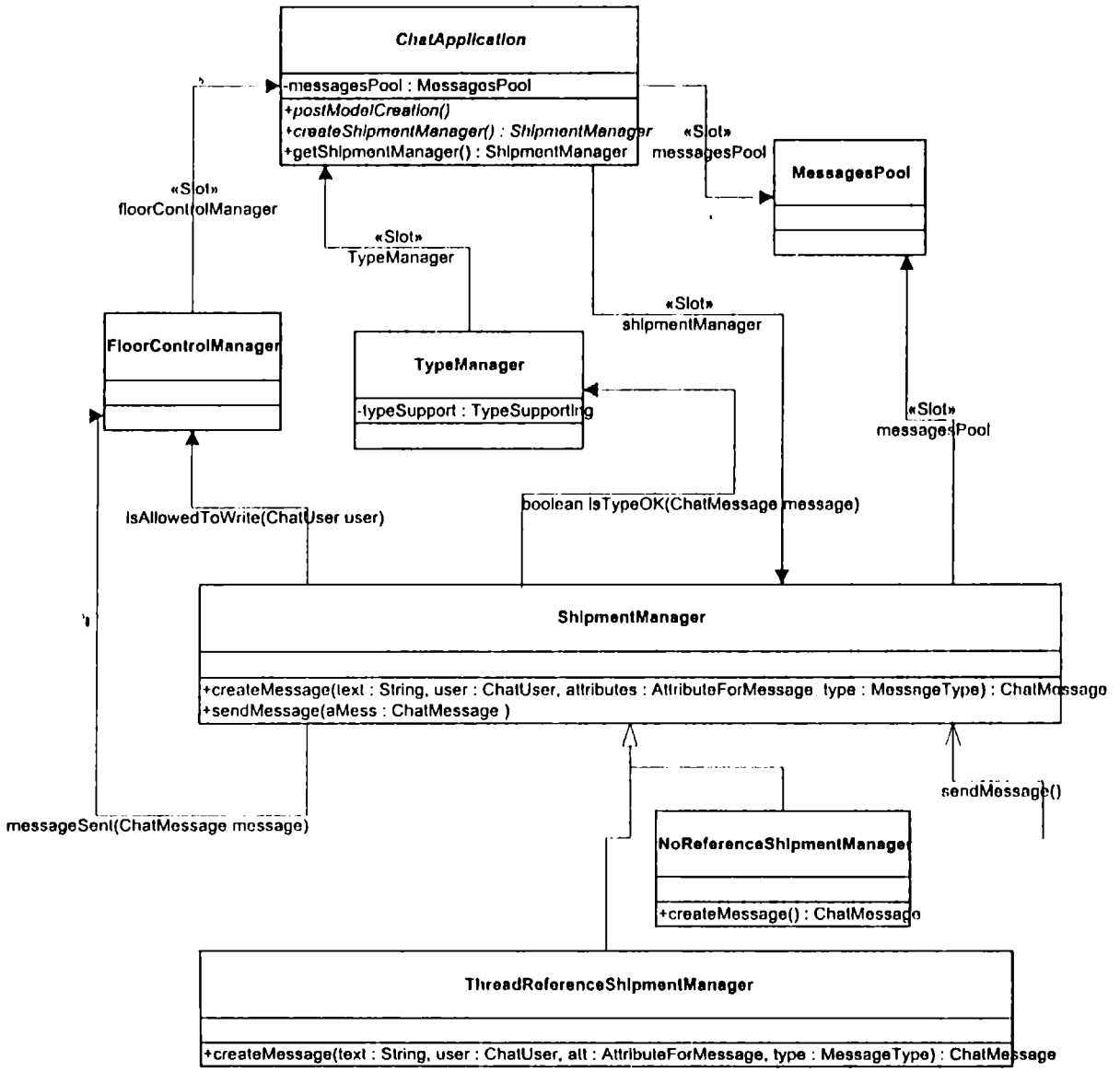


Diagrama B.5 ShipmentManager

En este diagrama pueden verse también las dos subclases concretas de ShipmentManager, cuando el sistema trabaje directamente con instancias de ChatMessage, entonces puede emplearse como shipment manager el NoReferenceShipmentManager. Si en cambio se requieren mensajes mas personalizados, o que incluyan otro tipo de información, entonces se puede subclasificar ShipmentManager para redefinir el método *createMessage()* como más sea conveniente. Si es necesario crear o modificar las restricciones necesarias para el envió de nuevos mensajes, basta con redefinir el método *sendMessage()*.

FloorControlManager

El FloorControlManager es el corazón de cualquier componente Chatblocks. Define el protocolo de comunicación que utilizará el sistema. El FloorControlManager colabora con el ShipmentManager para decidir cuando un usuario puede enviar un mensaje. Necesita interactuar con el UsersManager y con el RoleManager, debido a que de acuerdo a los usuarios en línea, se puede alterar el flujo de la conversación.

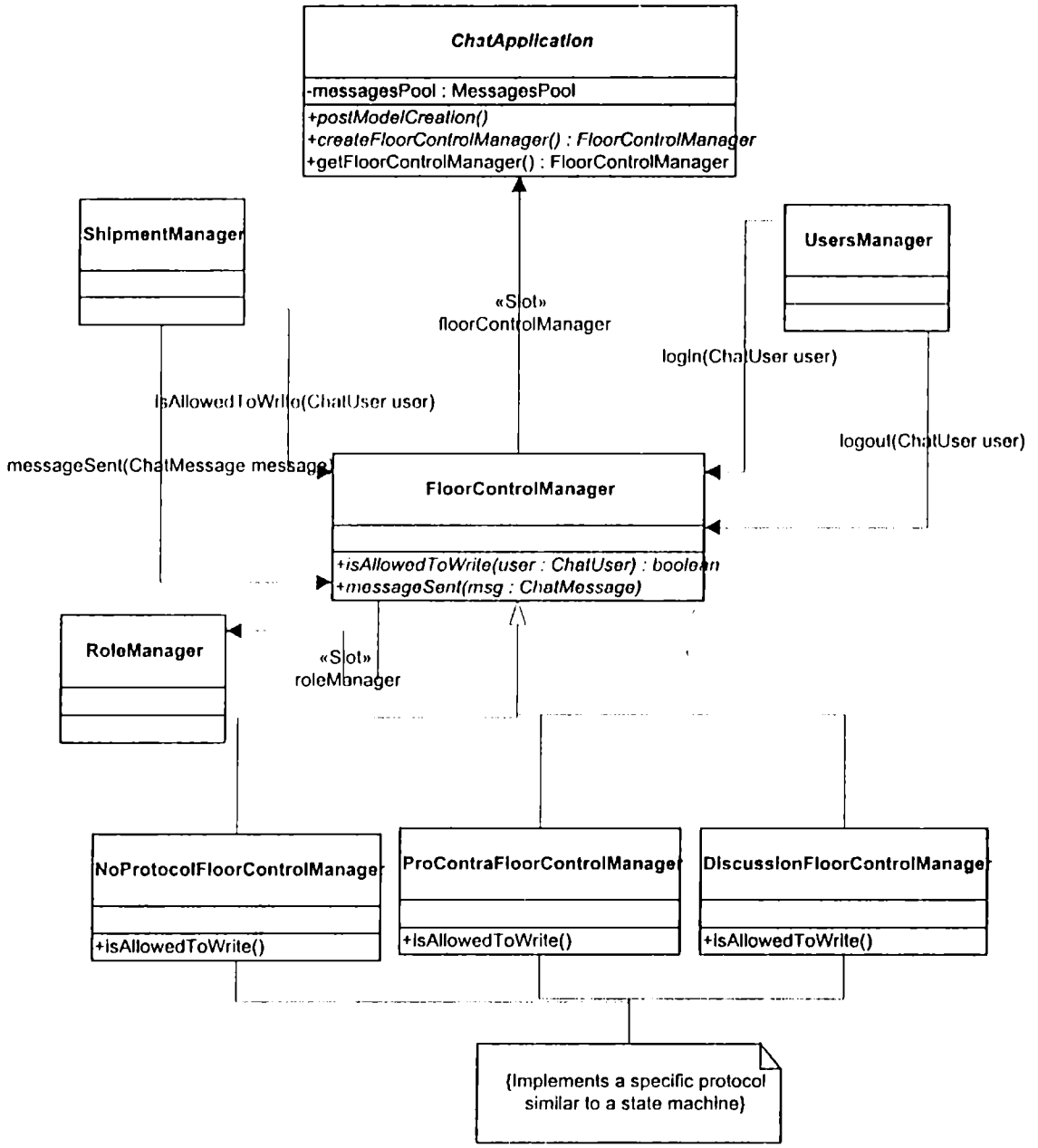


Diagrama B.6 FloorControlManager

En el diagrama también se pueden observar algunas de las clases concretas de FloorControlManager. La clase NoProtocolFlooControlManager no implementa ningún protocolo en particular, simplemente responde siempre afirmativamente al mensaje isAllowedToWrite. Esta clase es útil cuando simplemente se quiere construir un chat room que funcione de forma similar a los tradicionales.

ReceptionManager

El ReceptionManager es empleado por los view para recuperar desde el MessagePool los mensajes enviados. Este bloque puede emplearse para la aplicación de filtros generales, es decir, aplicables para todos los usuarios.

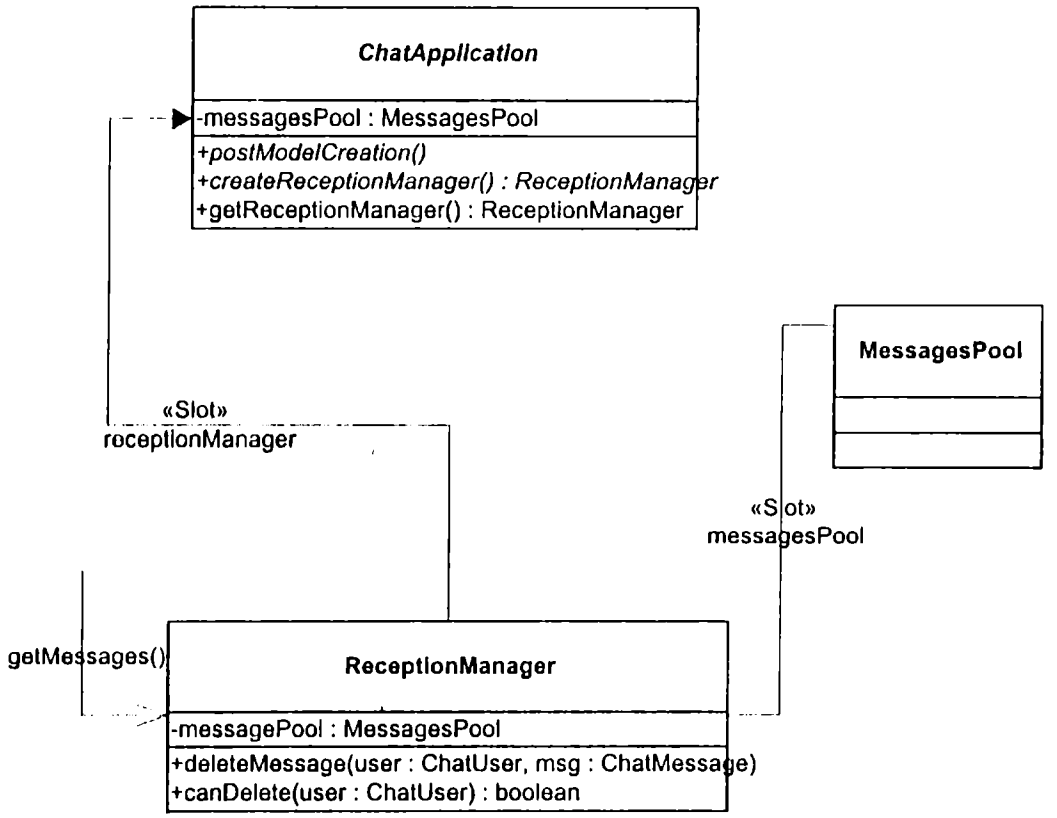


Diagrama B.7 ReceptionManager

En el ReceptionManager pueden incluirse restricciones tales como la decisión de que usuarios pueden eliminar o editar los mensajes enviados. En esta implementación, solo el autor o un usuario administrador esta habilitado para eliminar un mensaje del MessagePool.

UsersActivityView

El UsersAcitvityView muestra la lista de participantes y refleja el estado actual de cada uno. Para ello se compone de un conjunto de UserGroupListView. Cada UserGroupList puede ser personalizado y configurado para que muestre un subconjunto de los usuarios en el orden y apariencia deseados.

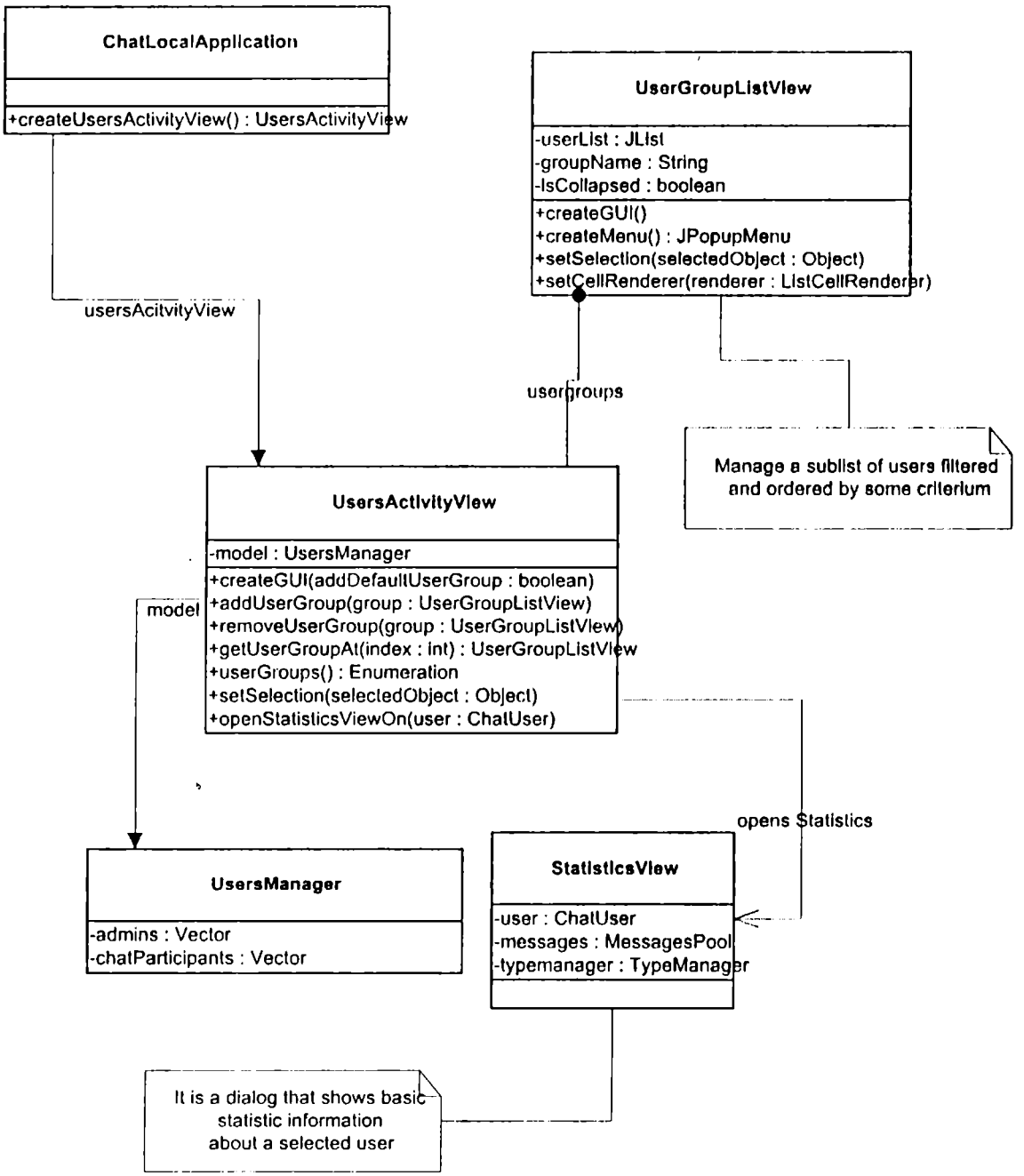


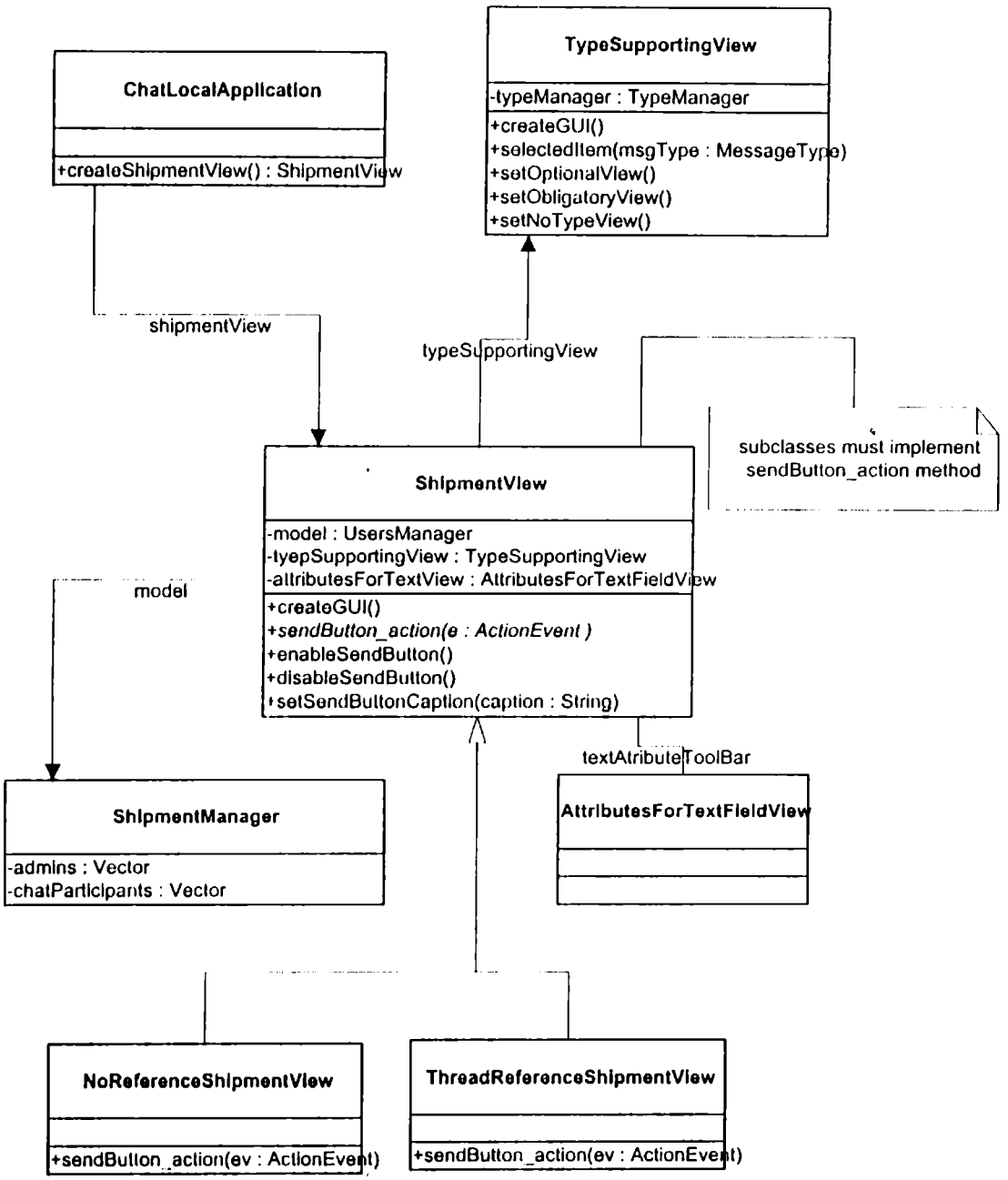
Diagrama B.8 UsersActivityView

Los UserGroupListView pueden ser agregados o quitados dinámicamente del UsersActivityView, de este modo se obtiene una lista de usuarios totalmente configurable y personalizable. Cada UserGroupListView emplea un ListCellRenderer para decidir como mostrar visualmente la lista de usuarios que le sirve de modelo.
Como otra función complementaria, el UsersActivityView es capaz de abrir un diálogo informativo acerca de las operaciones hechas por un determinado usuario. La clase

StatisticsView muestra información básica sobre la cantidad de mensajes enviados de cada tipo.

ShipmentView

El ShipmentView ofrece los controles visuales necesarios para enviar un mensaje al MessagePool. El TypeSupportingView se configurará de acuerdo a los tipos permitidos y al grado de obligatoriedad de los mismos



ramas del Diseño del Framework

Diagrama B.9 ShipmentView

Las subclases de ShipmentView deben implementar el método:
sendButton_action (ActionEvent ev)
mediante el cual el ShipmentView, crea y configura un nuevo mensaje (una subclase de ChatMessage) y lo envía a su modelo (el ShipmentManager)

FloorControlView

El FloorControlView contiene los controles visuales para que el usuario pueda llevar a cabo la conversación, de acuerdo al protocolo de comunicación del sistema. Es decir, si el protocolo requiere que el usuario para poder enviar un mensaje primero debe solicitar el turno, entonces el botón o comando destinado a ejecutar la petición del turno deberá estar presente en el FloorControlView. El método createGUI() es redefinido en cada una de las subclases precisamente para personalizar el bloque visual con los controles específicos.

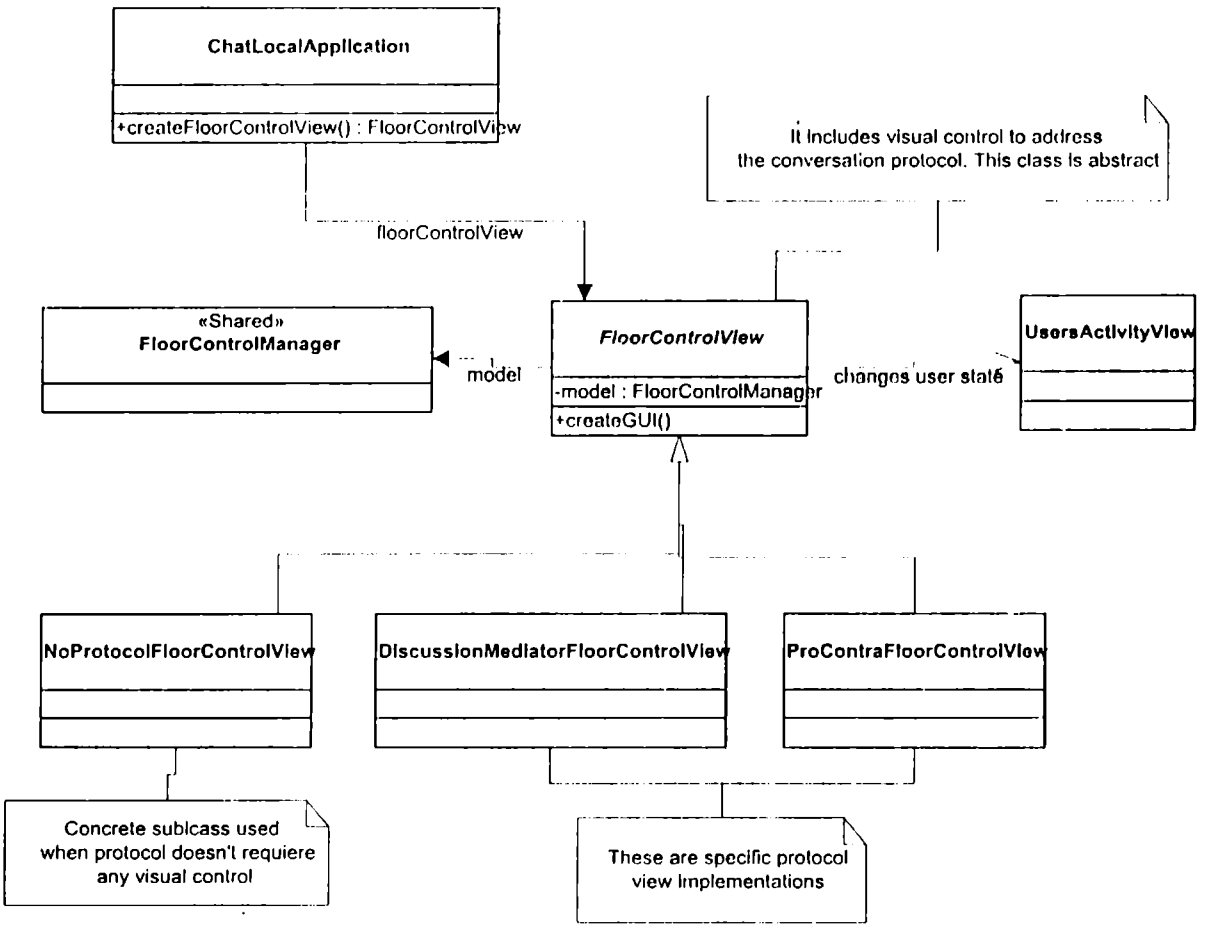


Diagrama B.10 FloorControlView

La ejecución de ciertos comandos presentes en el FloorControlView puede provocar cambios de estado en los usuarios, tales cambios serán reflejados por el

UsersActivityView. En algunas situaciones puede ser necesaria grado de cooperación mayor entre el UsersActivityView y el FloorControlView, ya que podría emplearse la misma lista de usuarios para reflejar el estado actual del protocolo (por ejemplo ver por medio de diferentes UserGroupListView que usuarios están esperando algún evento, etc.).

ReceptionView

El ReceptionView presenta los mensajes enviados al usuario de diferentes formas de acuerdo a los MessageViewer con los que esté configurado. Ofrece la posibilidad de incorporar o cambiar las vistas posibles para los mensajes en tiempo de ejecución simplemente agregando o quitando objetos que implementen la interfaz MessageViewer.

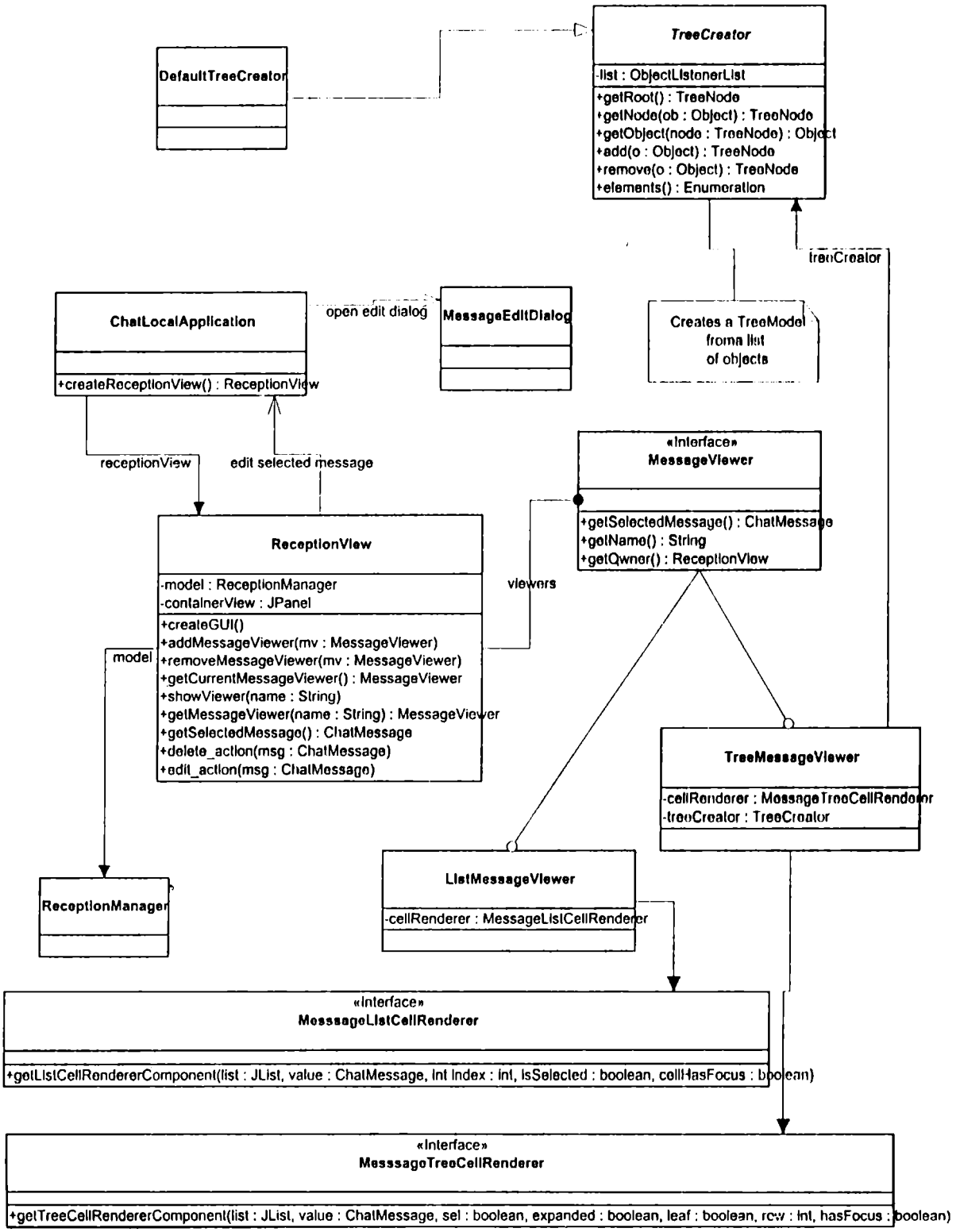


Diagrama B.11 ReceptionView

Chatblocks implementa dos clases diseñadas para poder visualizar los mensajes en forma de lista, y en forma de árbol (útil en aplicaciones como el ThreadChat, véase el apéndice Experiencias de Uso): ListMessageViewer y TreeMessageViewer.
Ambas clases ofrecen la posibilidad al programador de personalizar o definir sus propios renderers. El TreeMessageViewer permite la visualización de una estructura de árbol, para ello emplea un TreeCreator, el cual es el objeto que debe implementar el algoritmo que a partir de una lista (el MessagePool solo contiene una lista de los mensajes enviados), construya un árbol. Por defecto TreeMessageViewer emplea un DefaultTreeCreator.

Envío de contribuciones

En los diagramas siguientes puede observarse la cadena de métodos disparados por el envío de una nueva contribución de un usuario.

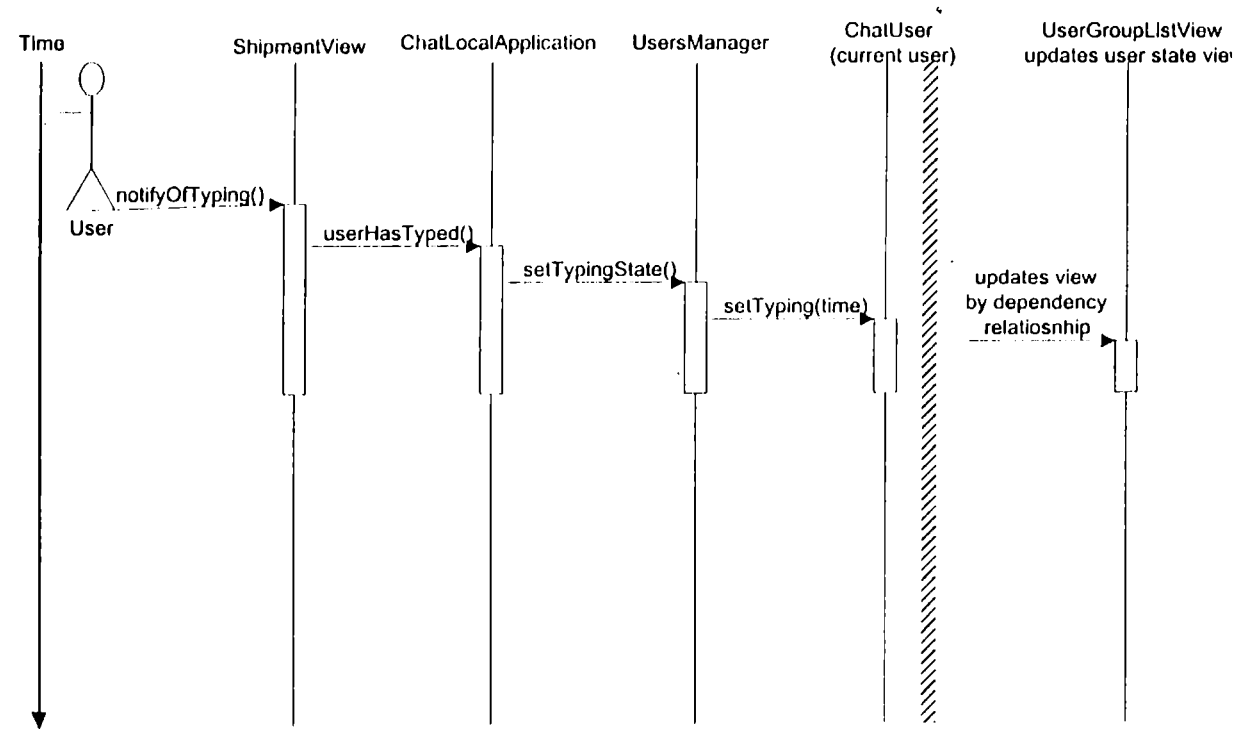


Diagrama B.12 Envío de un mensaje: el usuario tipea un carácter sobre un campo de texto en el ShipmentView

Cuando el usuario comienza a escribir caracteres se produce una serie de mensajes que terminan alterando el estado actual del usuario de ACTIVE a TYPING, este cambio de estado es propagado por el mecanismo de dependencias hasta el componente visual encargado de reflejar en la lista de usuarios (mediante un icono para estado) dicho cambio. De este modo se obtiene un mayor awareness de la actividad que esta realizando cada usuario. Basta con observar la lista de usuarios para saber si otro usuario esta escribiendo algo, o simplemente esta a la espera de los demás.

Una vez que el usuario ha terminado de escribir su mensaje deberá activar el control de acción correspondiente para enviar el mensaje (el control de acción por defecto incluido en el ShipmentView es el botón con la etiqueta SEND, o alternativamente presionar las teclas CTRL. Y ENTER).

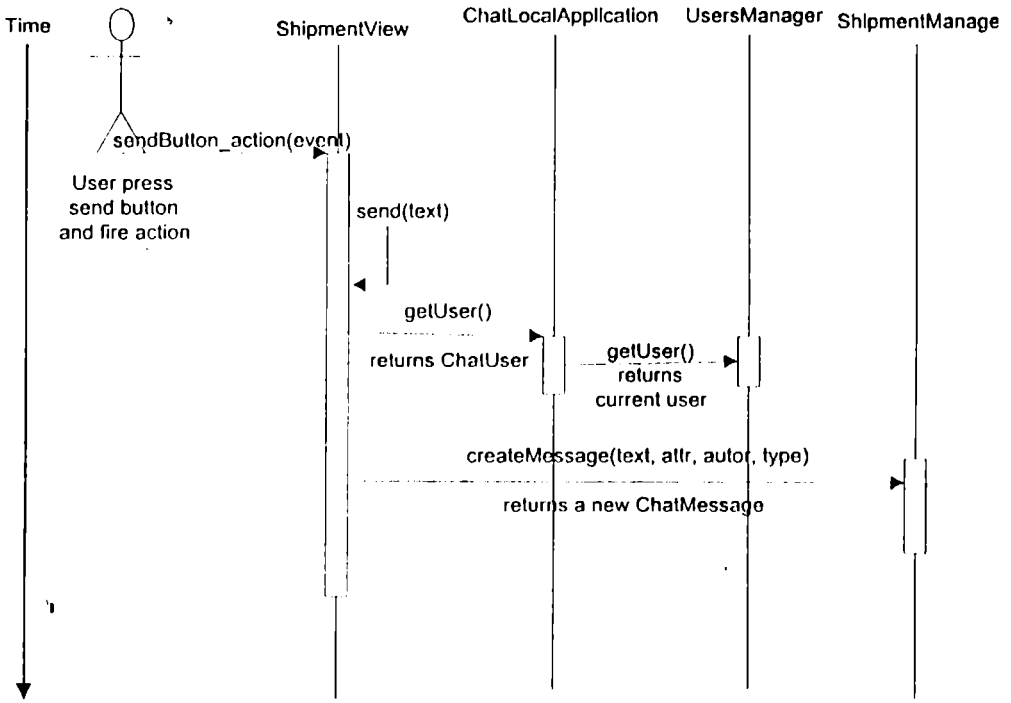


Diagrama B.13 Envío de un mensaje: el usuario dispara la acción de enviar, el ShipmentView solicita al ShipmentManager un mensaje nuevo con el texto ingresado por el usuario

El ShipmenteManager es el objeto que conoce como crear objetos ChatMessage. EL ShipmentView solicita un nuevo ChatMessage enviando la información referente al texto, autor y tipo de mensaje al ShipmentManager.

Por defecto, el ShipmentManager verifica tres condiciones antes de proceder a enviar el mensaje:

- que el autor del mensaje esté habilitado para enviar mensajes, es decir, que es aceptado por el FloorControlManager;
- que el tipo asignado al nuevo mensaje sea correcto, es decir, es un tipo permitido por el TypeManager;
- que el contenido del mensaje (el texto) sea válido, por ejemplo verificar que el texto no sea nulo.
-

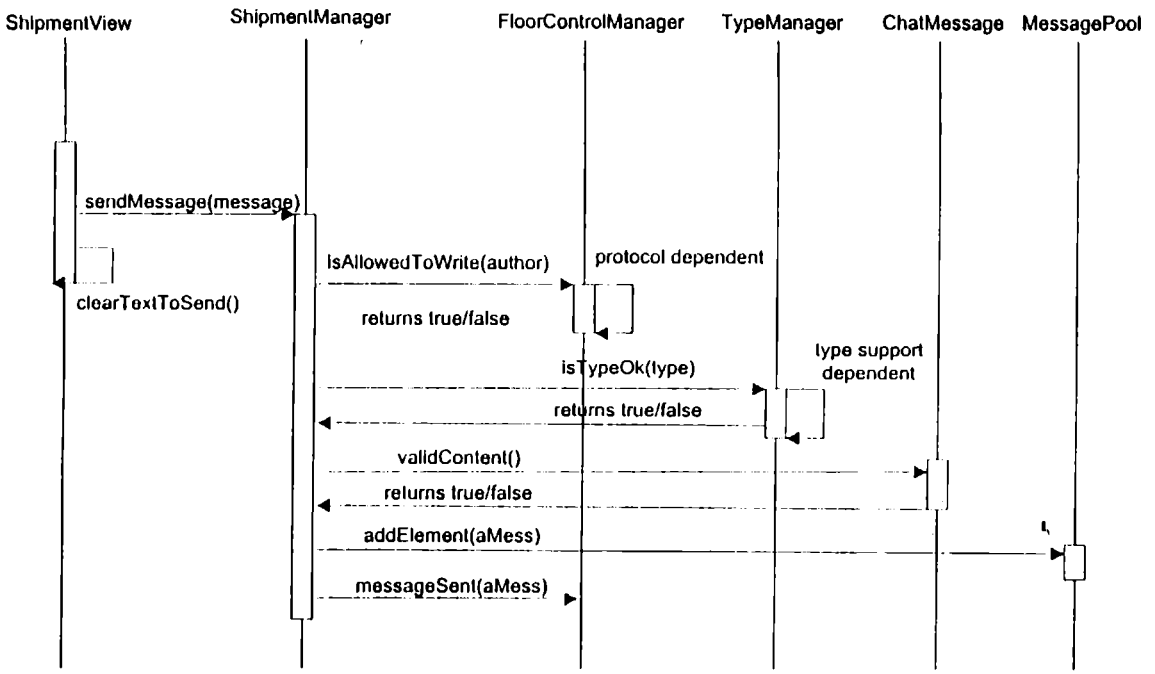


Diagrama B.14 Envío de mensajes: verificación de condiciones de envío y notificación del envío realizado

Si todas las condiciones se cumplen, el ShipmentManager envía el mensaje y avisa del evento al FloorControlManager ya que podría verse alterado el flujo de la conversación. Luego a través del mecanismo de dependencias el ReceptionView es notificado de que nuevo elemento ha sido agregado al MessagesPool y por lo tanto debe actualizar la vista de los datos.

B.5 Construcción de Componentes Chatblocks

En esta sección puede observarse la configuración de los bloques locales y del modelo compartido de las aplicaciones construidas con Chatblocks.

Debido a que estos componentes forman una familia de aplicaciones, todas cuentan con estructuras similares sobre todo la representación visual de cada una. El componente más singular es el Snapchat que por sus características debería merece un trato especial

SimpleChat

Es el más simple de los sistemas, no implementa ningún protocolo específico, en esta versión se incluye un soporte no obligatorio de tipos

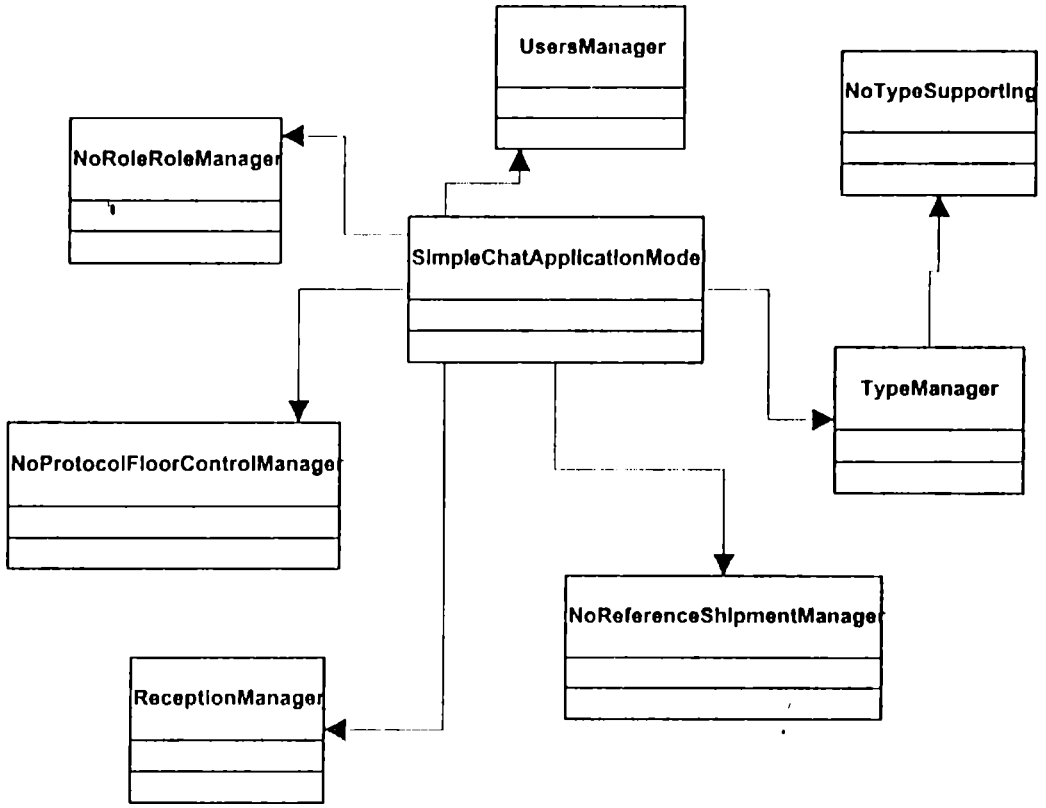


Diagrama B.15 SimpleChat Modelo Compartido

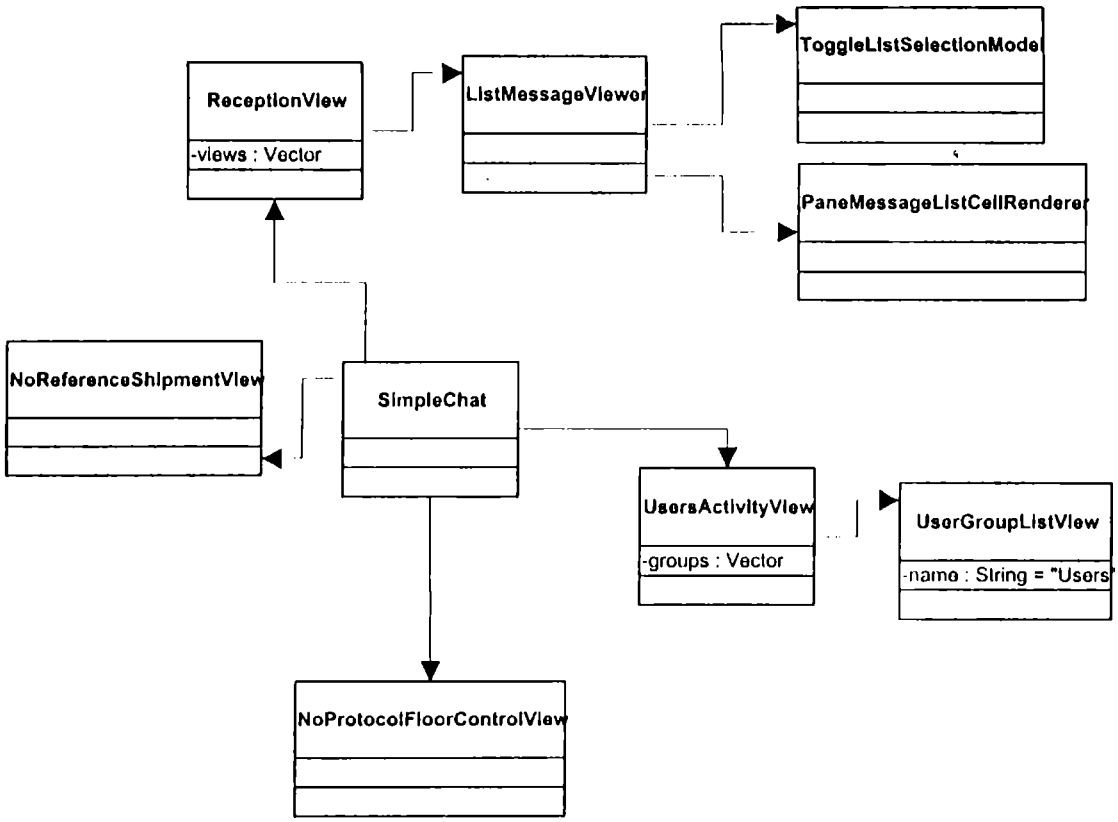


Diagrama B.16 SimpleChat Modelo Local

Pro-Contra

La única diferencia con el anterior reside en el empleo de un protocolo mas definido y explicito de comunicación que no permite enviar mas de un mensaje sin antes recibir una respuesta por parte del otro participante.

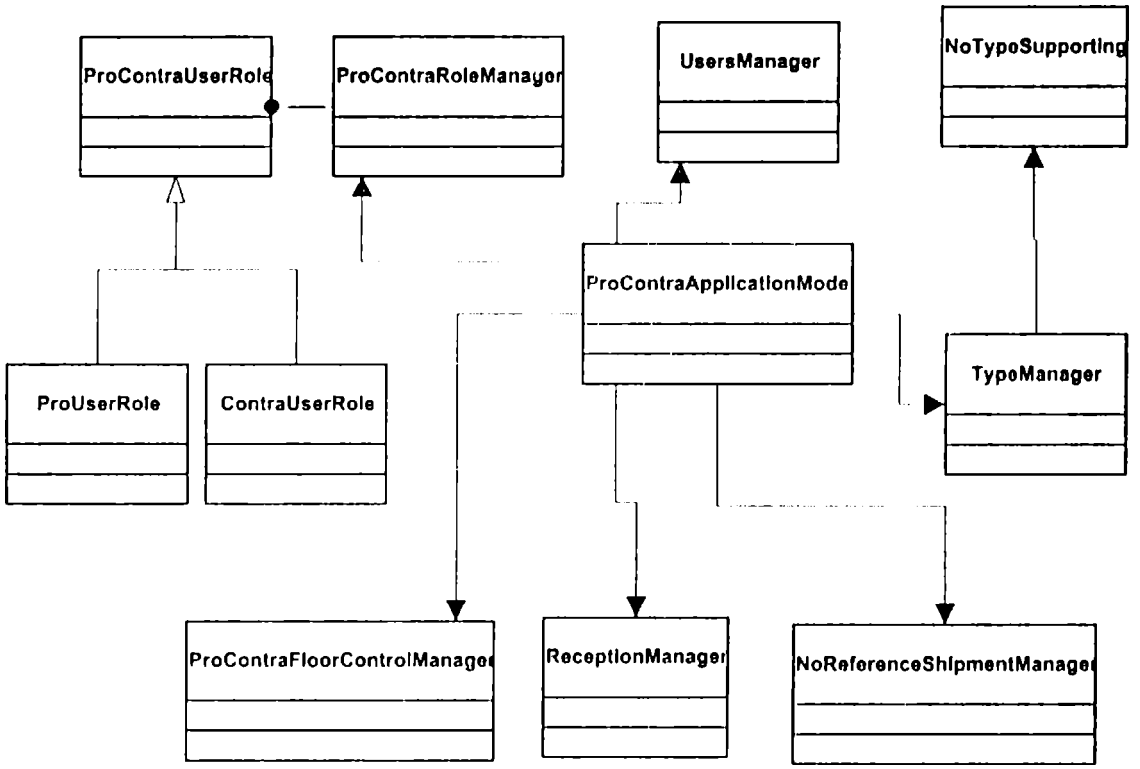


Diagrama B.17 ProContra Modelo Compartido

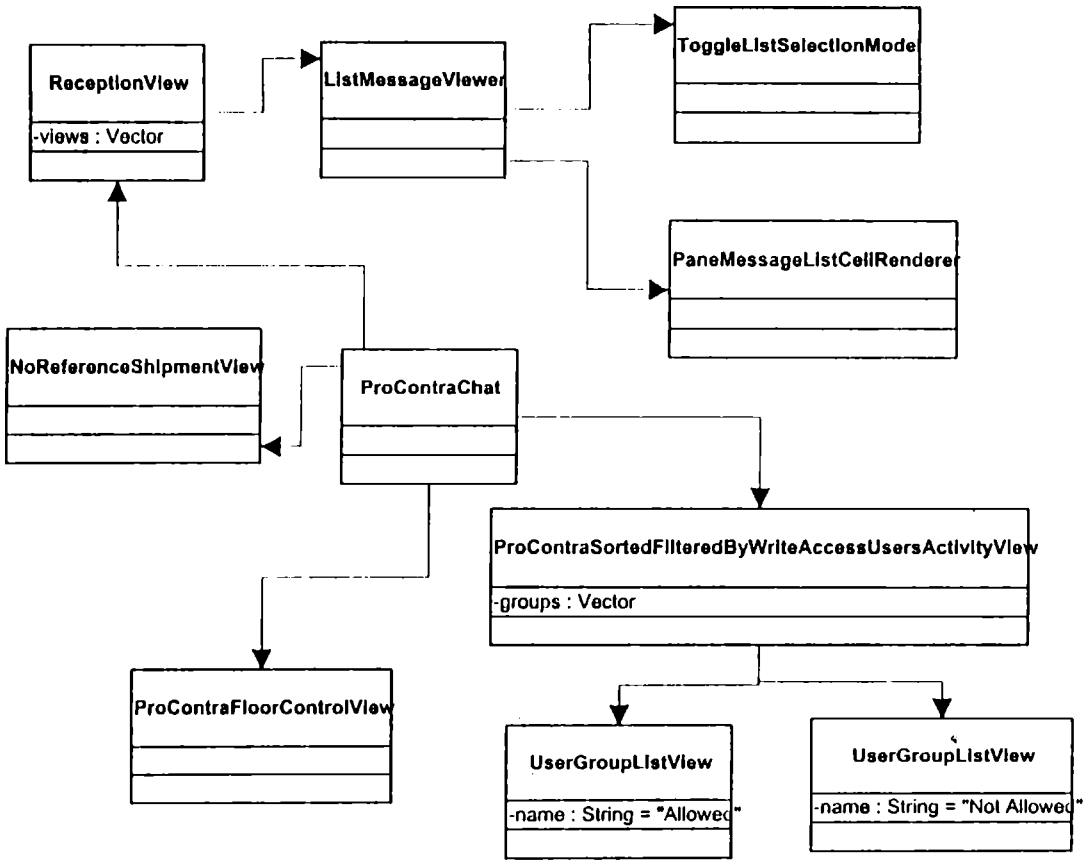


Diagrama B.18 ProContra Modelo Local

Human Mediated Discussion y Discussion with Auto-tutor

Estos dos componentes comparten casi todos los bloques, la diferencia reside en el floor control manager: el auto tutor se encarga de asignar automáticamente al próximo usuario que tiene permiso de enviar un mensaje, mientras que el human mediated se apoya en un tercer participante para asignar los turnos.

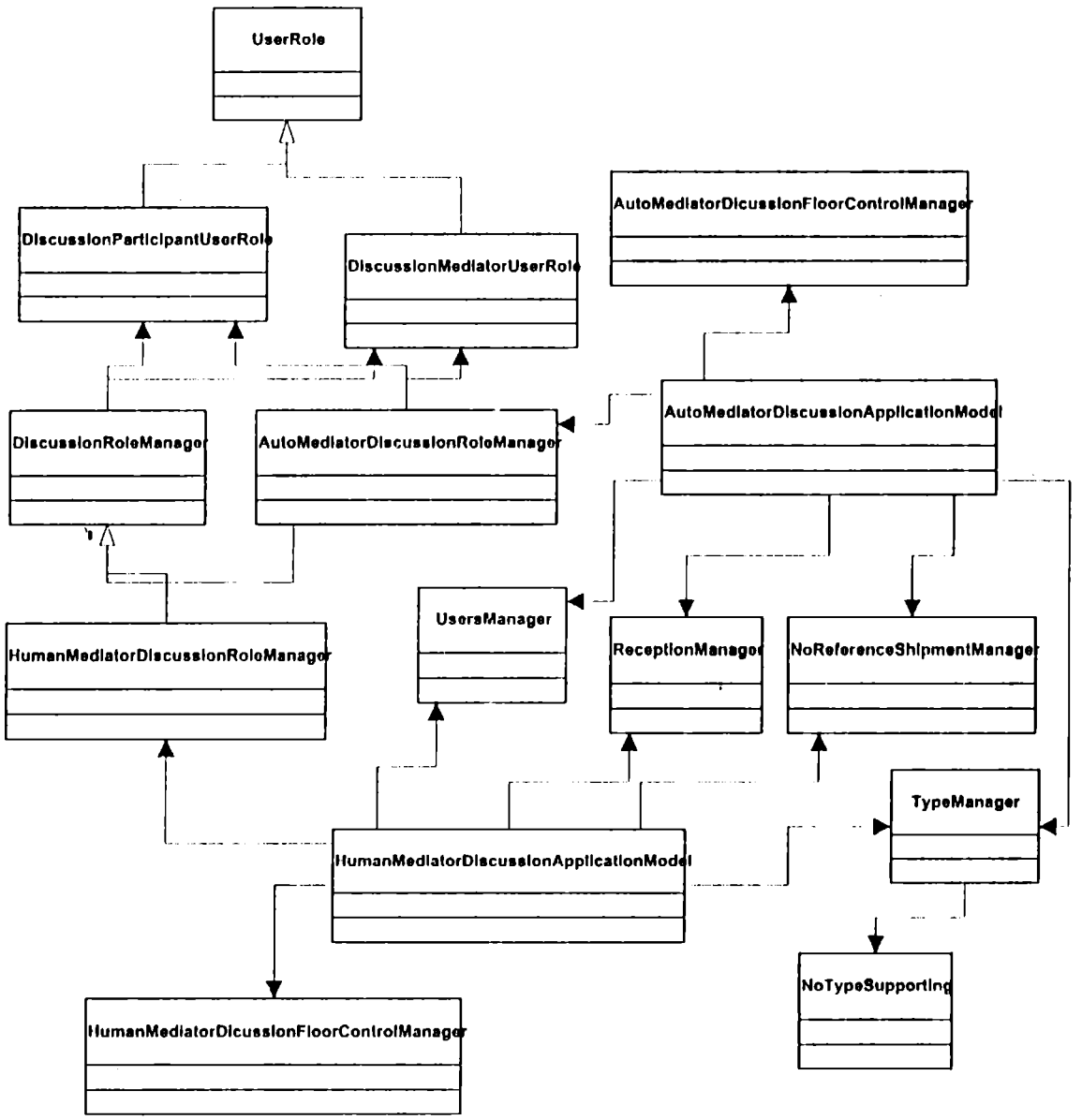


Diagrama B.19 Modelo Compartido de Human Mediated Discusión y Discusión with Auto-Tutor

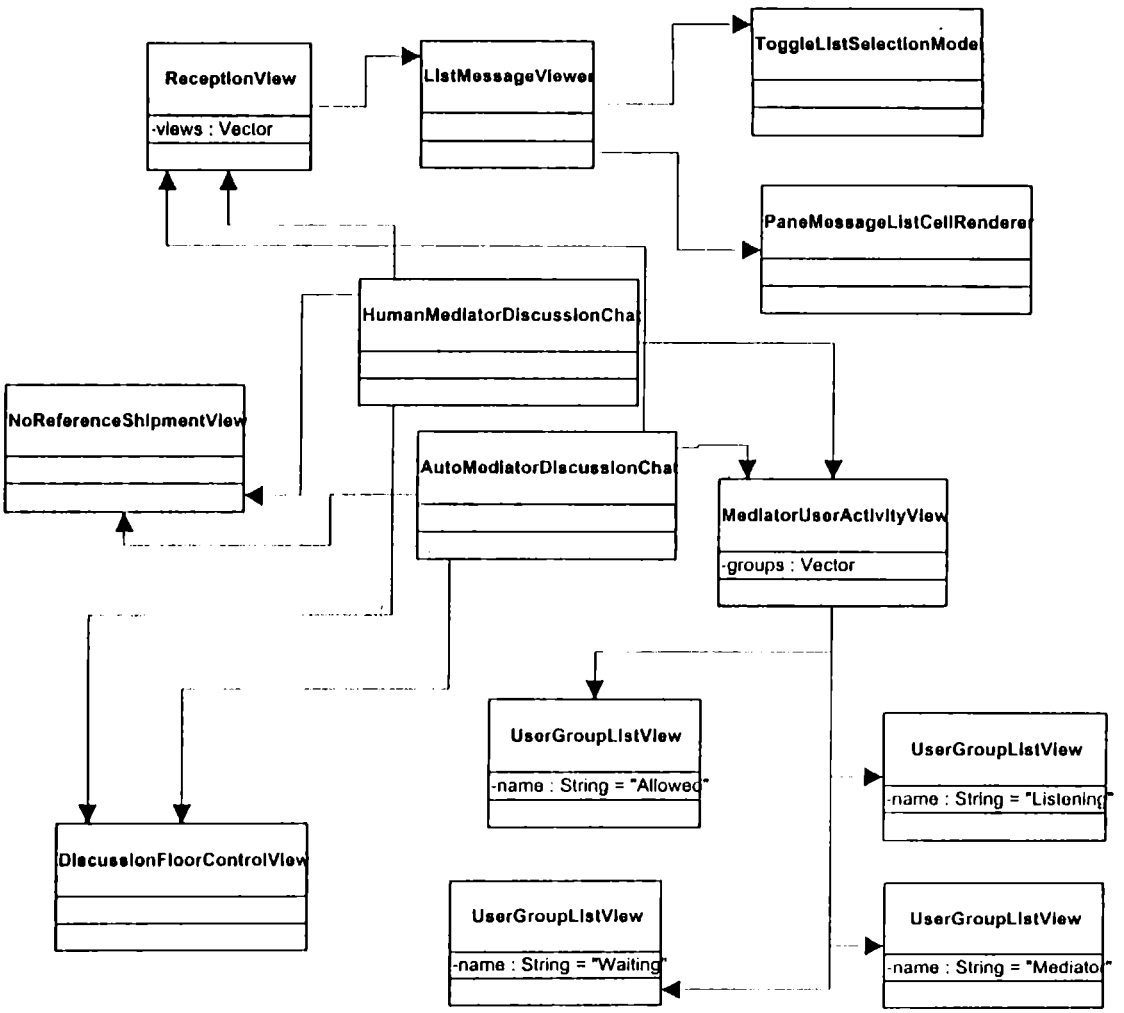


Diagrama B.20 Modelo Local de Human Mediated Discusión y Discusión with Auto-Tutor

Threaded Chat

La aplicación es similar al SimpleChat, pero ofreciendo una visualización en forma de árbol de los mensajes enviados (ReceptionView). El usuario puede contestar a un mensaje enviado, seleccionándolo del árbol y enviando un mensaje con referencia a éste. De este modo se arman threads de conversación.

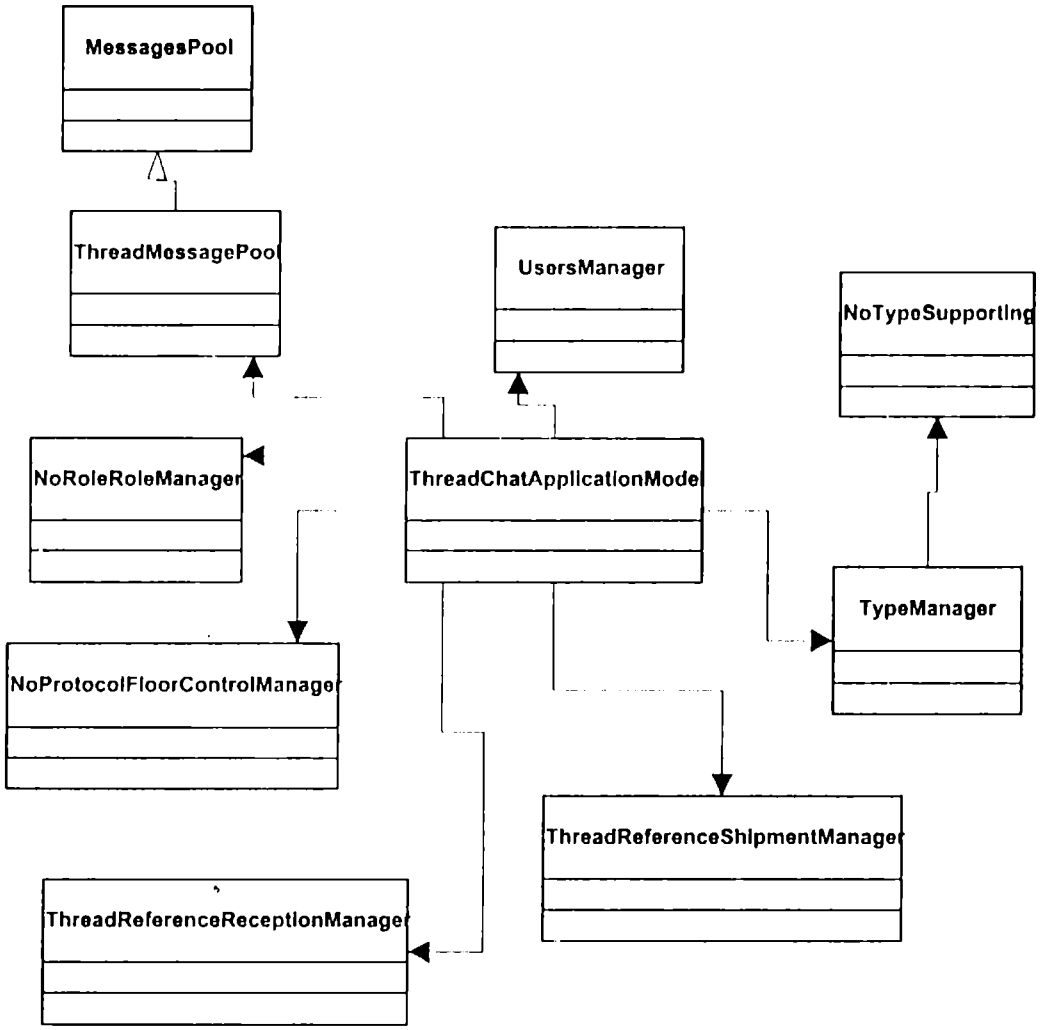


Diagrama B.21 ThreadChat Modelo Compartido

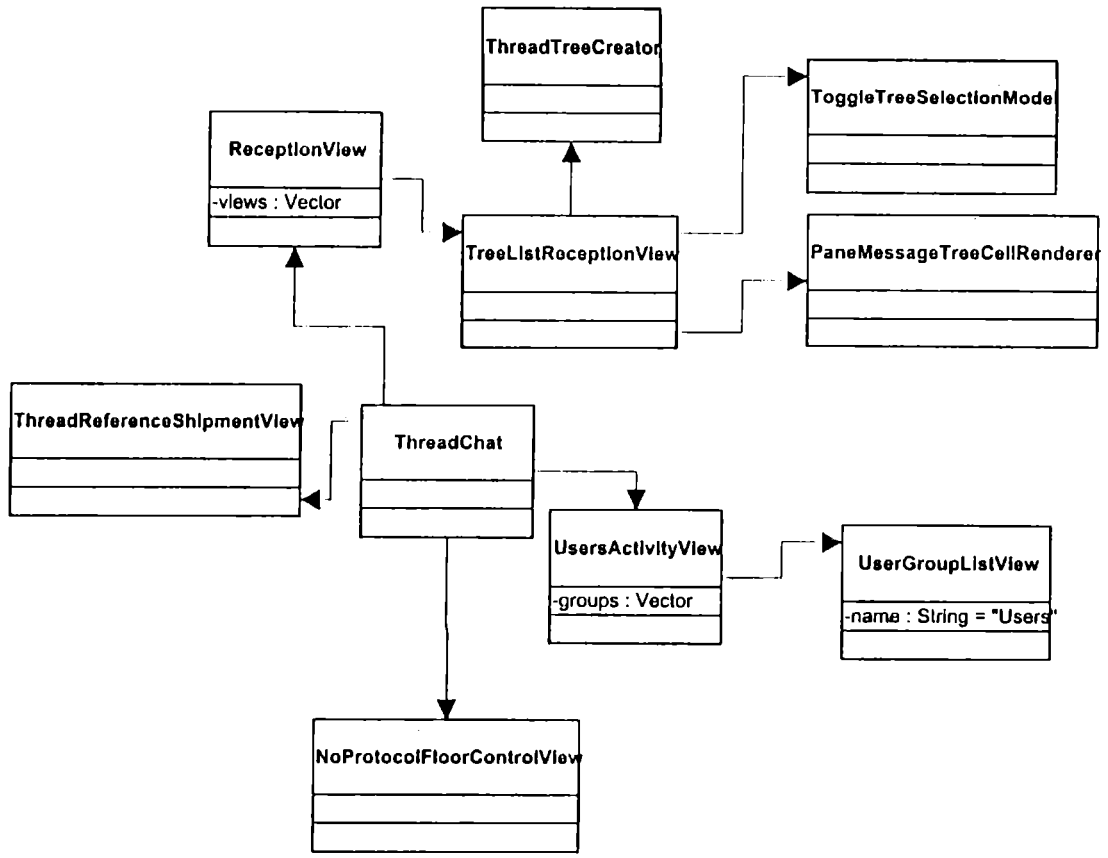


Diagrama B.22 ThreadChat Modelo Local

Learning Protocol Chat

La principal diferencia reside en el protocolo de conversación: intenta simular un cuarto de clases virtual, donde existe un profesor (ExpertRole) y alumnos (LearnerRole). Los alumnos preguntan y el profesor responde. Para marcar la diferencia entre preguntar, responder, o simplemente comentar se emplea el uso de tipos obligatorios (las subclases de LearningProtocolMessageType representan los tres tipos mencionados: pregunta, respuesta y comentario)

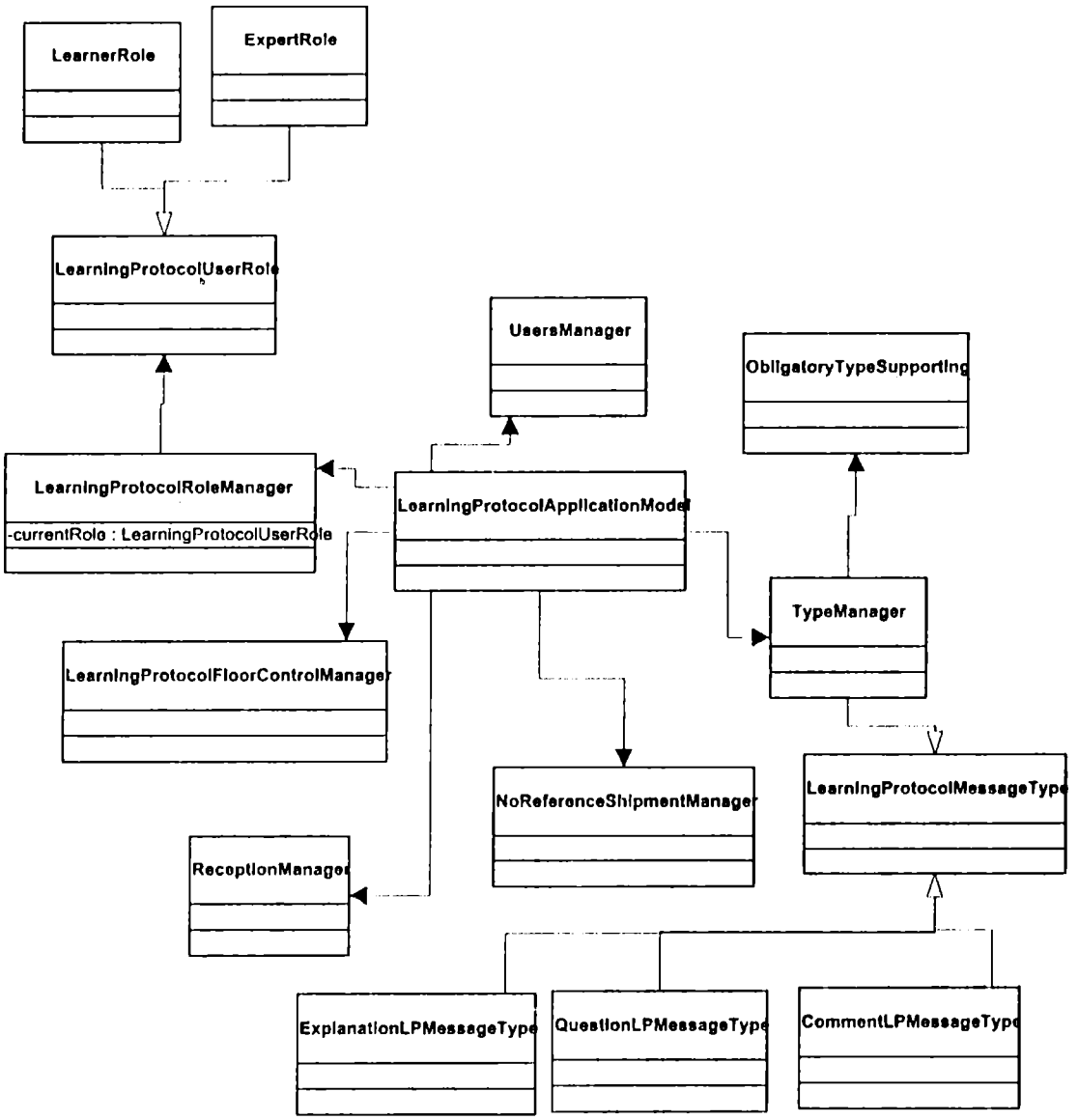


Diagrama B.23 Modelo Compartido de Learning Protocol Chat

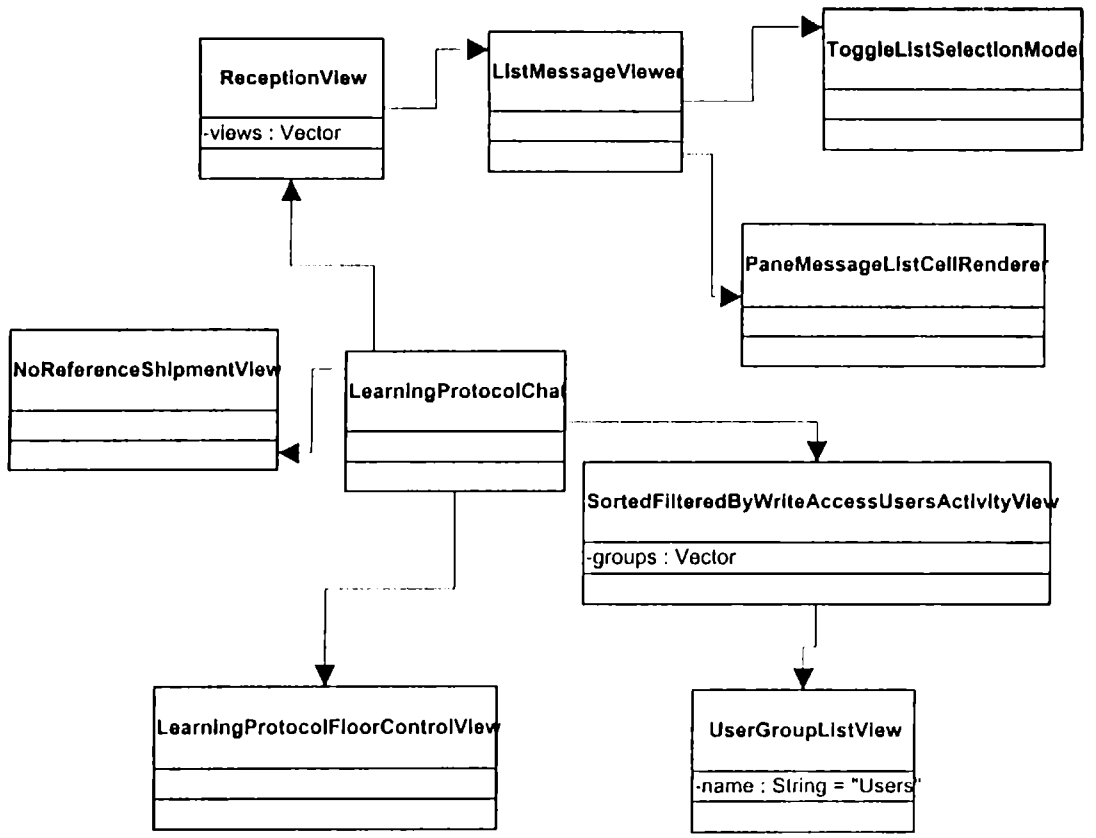


Diagrama B.24 Modelo Local de Learning Protocol Chat

Snapchat

Es la mas compleja de las aplicaciones construida con Chatblocks, se destaca por en ofrecer a los usuarios una pizarra de dibujo y edición de distintas clases de documentos. Dicha pizarra es modelada por la clase DocumentViewer. El usuario envía nuevos mensajes simplemente marcando alguna parte de un documento y anexándole su comentario. El mensaje es visualizado por el resto de los usuarios en la lista presentada por el ReceptionView, basta con seleccionar un mensaje de la lista para que la aplicación trace una referencia hasta el lugar del documento al que corresponde dicho mensaje.

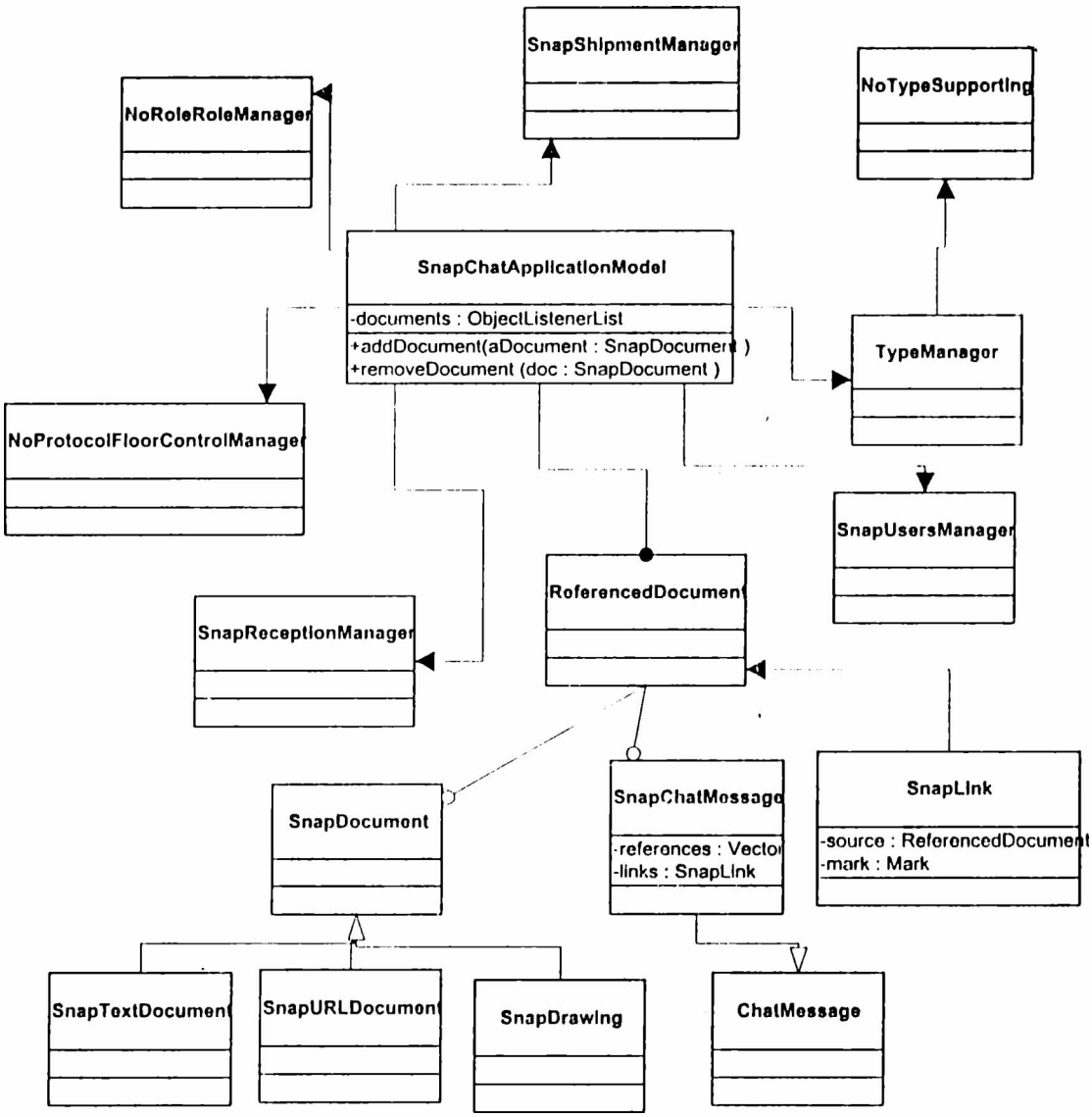


Diagrama B.25 Modelo Compartido de Snapchat

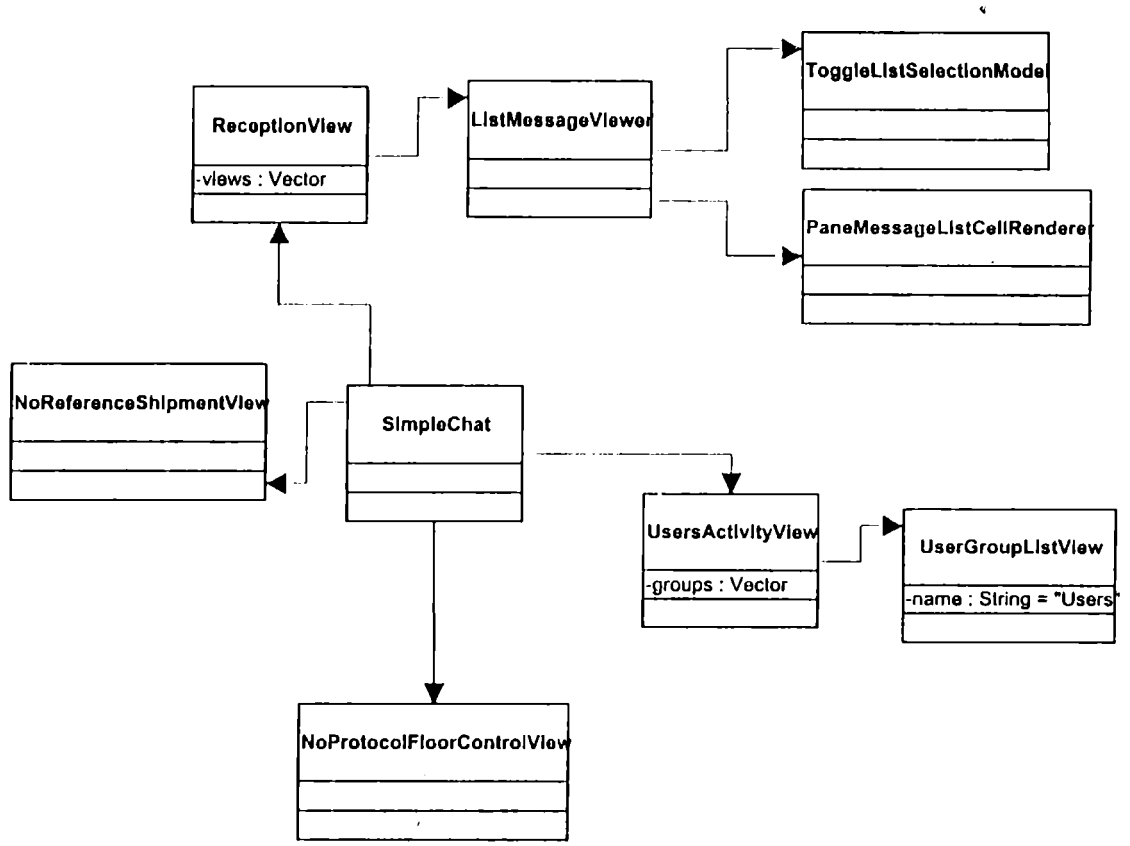


Diagrama B.26 Modelo Compartido de Snapchat

En este diagrama solo se muestra una parte de los modelos compartidos y locales, esencialmente lo que tiene que ver con Chatblocks.

B.6 Diagramas de Notación de Protocolos

De acuerdo a una notación para representar formalmente los protocolos de comunicación que aún se encuentra en desarrollo[1], podemos visualizar gráficamente cada uno de los protocolos de los componentes que han sido desarrollados con Chatblocks con los diagramas siguientes.

La idea básica es graficar los roles y los tipos de contribuciones que éstos pueden hacer indicando con una flecha a quien le corresponde el turno tras el envío de una nueva contribución:

[1] Vea la sección Trabajos Futuros del Capítulo 8 para mas detalles sobre notación formal para protocolos de comunicación

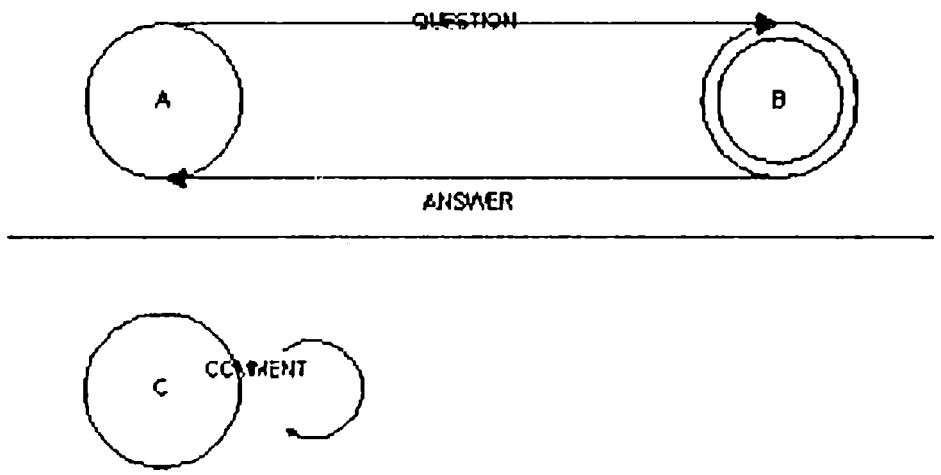


Figura B.1 Idea básica de la representación gráfica de un protocolo de comunicación

Siguiendo esta idea, podemos representar a dos de los protocolos mas simple de la siguiente forma:

El SimpleChat esta compuesto por un único rol que puede enviar en cualquier momento, sin restricciones:

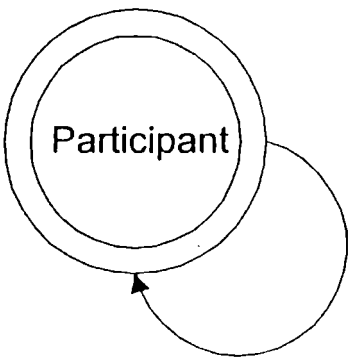


Figura B.2 Notación gráfica del protocolo de comunicación del SimpleChat

En el Pro-Contra existen dos roles, cuando uno de los roles envía un mensaje habilita a su opuesto a enviar su respuesta.

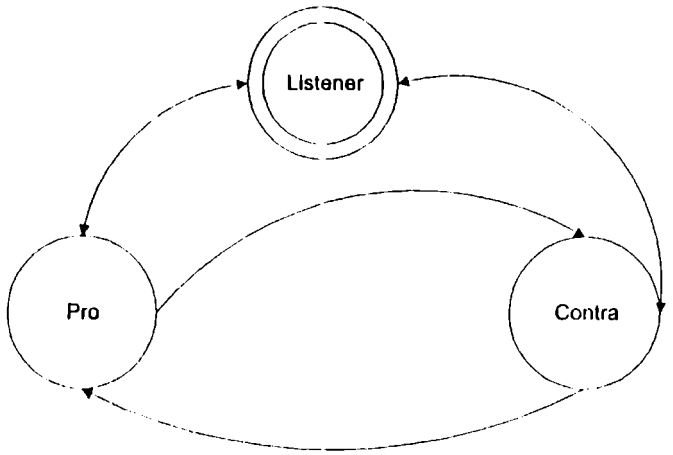


Figura B.3 Notación gráfica del protocolo de comunicación del Pro-Contra Chat

En este diagrama se incluye un tercer rol formado por aquellos usuario que solo están atendiendo la conversación. Además estos usuarios, de acuerdo al gráfico, podrían enviar mensajes durante cualquier momento de la conversación.

No expongo aquí las figuras correspondientes al resto de los componentes Chatblocks debido a que es necesario profundizar mucho mas acerca de esta notación para poder explicar los diagramas de protocolos mas complejos como el Learning Protocol, y no es propósito de esta tesis involucrarse en tal tema.

Listado de Ilustraciones

Tablas

Figuras

Diagramas

Bibliografía

[Col97] David Coleman. Groupware - Collaborative Strategies for Corporate LANs and Intranets. Prentice-Hall, 1997. ISBN: 0-13-727728-8.

[Gre91] Saul Greenberg. Computer-supported Cooperative Work and Groupware. Academic Press, London, 1991.

[DCS94] Prasun Dewan, Rajiv Choudhary, and Honghai Shen. An Editing-Based Characterization of the Design Space of Collaborative Applications. Journal of Organizational Computing, 1994.

[The01] The Yankee Group. Communication, Collaboration, Coordination: The "Three Cs" of Workgroup Computing. www.yankeegroup.com, 2001.

[Cha96] David Chappell. Understanding ActiveX and OLE. Microsoft Press, 1996. ISBN: 1-57231-216-5.

[SKSH96] C. Schuckmann, L. Kirchner, J. Schümmer, and J.M. Haake. Designing Object-Oriented synchronous groupware with COAST. In Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96), pages 30{38. ACM Press, New York, 1996.

[SSS99] Christian Schuckmann, Jan Schümmer, and Peter Seitz. Modeling Collaboration using Shared Objects. In Proceedings of ACM GROUP99, International Conference on Supporting Group Work. ACM Press, 1999.

[VC99] F. Viegas, J. Donath, K. Karahalios. "Visualizing Conversation". MIT Media Lab. 1999.

[THDIS97] J. Hewitt. Beyond Threaded Discourse. WebNet'97.

[JOFO91] R. Johnson, B. Foote. Designing Reusable Classes. OOPSLA'91.

[FASCH97] M. Fayad, D. Schmidt. "Object Oriented Application Frameworks". Communications of the ACM Octubre 1997/Vol. 40, No 10, page 32.

[JOACM97] R. Johnson. Frameworks = Componets + Patterns. Communications of the ACM Octubre 1997/Vol. 40, No 10, page 39.

[VSD99] D. Vronay, M. Smith, S. Drucker. Alternative Interfaces for Chat. CHI'99.

[VIDO99] F. Viegas, J. Donath. Chat Circles. CHI'99.

[VIDOKA99] F. Viegas, J. Donath, K. Karahalios. "Visualizing Conversation". MIT Media Lab. 1999.

[HEW97] J. Hewitt. Beyond Threaded Discourse. WebNet'97.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vessides. Design Patterns: Elements for reusable Object-Oriented Programming. Addison-Wesley, 1995.

[SFJ99] Douglas C. Schmidt, Mohamed E. Fayad, Ralf E. Johnson. Building Application Frameworks. Wiley1999.

[BI93] Birrer, E.T. Frameworks in the financial engineering domain: An experience report. In Proceedings of ECOOP '93 Proceedings, Lecture Notes in Computer Science nr. 707, Springer-Verlag, 1993.

[CRHI] Campbell, R.H. and Islam, N. A technique for documenting the framework of an object-oriented system. Computing Systems

[FMEH] Fayad, M.E. and Hamu, D.S. Object-oriented enterprise frameworks: Make vs. buy decisions and guidelines for selection. Commun. ACM, submitted for publication.

[FMEH97] Fayad, M.E. and Hamu, D.S. Object-Oriented Enterprise Frameworks. Wiley, NY, 1997.

[HJE95] Hueni, H., Johnson, R., and Engel, R. A framework for network protocol software. In Proceedings of OOPSLA'95, (Austin, Texas, Oct.1995).

[PRE94] Prec, W. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1994.

[SCH97] Schmidt, D.C. Applying design patterns and frameworks to develop object-oriented communication software. In P. Salus, Ed., Handbook of Programming Languages, Volume I, MacMillan Computer Pub., 1997.

[DRAD91] Dringus, L., Adams, P. A Study of Delayed-Time and Real-Time Text-Based Computer-Mediated Communication Systems on Group Decision-Making Performance, Dissertation, Nova University, 1991

[EGR91] Ellis, C., Gibbs, S., and Rein, G., Groupware: Some Issues and Experiences, Communications of the ACM, Vol. 34, No. 1, 1991, 39-58

[ACKS96] Ackerman, M., Starr, B., Social Activity Indicators for Groupware, Computer, June 1996, 37-42

[ALTU98] Altun, A., Interaction Management Strategies on IRC and Virtual Chat Rooms, SITE 98 proceedings, March 10-14, 1998, 1223-1227

[BLK92] Beaudouin-Lafon, M., Karsenty, A., Transparency and Awareness in Real-Time Groupware Systems, UIST proceedings, 1992, 171-180

[DOPE92] Dourish, P., Bellotti, V., Awareness and Coordination in Shared Workspaces, CSCW proceedings, 1992, 107-114

[DOBLY92] Dourish, P., Bly, S., Portholes: Supporting Awareness in a Distributed Work Group, CHI proceedings, 1992, 541-547

[COAST] COAST: http://www.opencoast.org

Las herramientas de comunicación del tipo "chat", tales como los chat rooms de internet o los instant messengers han ido ganando una notoria popularidad entre los grupos de personas que requieren una comunicación rápida y fluida, sin necesidad de tener que cambiar de entorno de trabajo, con la posibilidad de compartir de forma inmediata el trabajo que estén realizando

Un sistema chat esta compuesto por una aplicación multiusuario, en donde dos o mas personas pueden intercambiar mensajes (en su modalidad mas simple, solo de texto) de forma sincrónica, en tiempo real.

Este tipo de herramientas surgieron casi con la misma aparición del correo electrónico, y con el correr de los años han ido ganando mas popularidad, sobre todo al surgir aplicaciones con interfaces mas intuitivas y fáciles de utilizar por el común de la gente. Sin embargo, aunque su aspecto se modificó bastante, aún siguen manteniendo la idea original, simplemente servir como un canal de comunicación sincrónico entre dos o mas personas, sin ningún tipo de control o regulación acerca del carácter y momento en que los usuarios pueden realizar sus aportes.

Pensando en estas falencias surgió Chatblocks, un framework que le brinda al desarrollador una forma rápida, totalmente extensible y personalizable de construir sistemas chat no convencionales en donde es la aplicación la que define la política y las reglas de cómo llevar adelante una conversación.

Universidad Nacional de La Plata,
Buenos Aires, Argentina

marzo de 2002