Trabajo de Grado.

Licenciatura en Informática.

Framework para la creación de chats.

Información del alumno

Nombre: Fernando García

Número de Alumno: 1535/5

Director

Dr. Gustavo H. Rossi

LIFIA. Universidad Nacional de La Plata. La Plata. Buenos Aires. Argentina.

Agradecimientos

Quisiera agradecer al Lic. Alejandro Fernández, ya que sin su ayuda no hubiese podido realizar esta tesis.

Indice

Capítulo 1

Introducción

A medida que la informática avanza, varias ramas de ella han crecido a pasos agigantados, entre ellas la de las comunicaciones por intermedio de la computadora, como ejemplos podemos citar: Tele conferencia, Audio conferencia, Pizarra de dibujo cooperativa.

Pero existe una herramienta de comunicación que no ha evolucionado a la par de las demás, el chat. Hoy en día, con una potencia de calculo 1000 veces superior a la de los años 70, el 90% de los chats son del tipo de los desarrollados en esa década. En el caso más simple se cuenta con una ventana de texto dividida en dos secciones, donde cada usuario escribe en una de ellas. Ambos pueden ver que esta escribiendo el otro, p.e.: Talk del S.O. UNIX.

En el caso de comunicación entre varios usuarios se utiliza la ventana de texto, donde todos los usuarios pueden agregar su aporte en cualquier momento y las veces que lo desee. Comunmente, las entradas se muestran en orden de llegada, e incluyen una referencia a su autor. Las aplicaciones chat más populares de la actualidad encajan en esta descripción eg: ICQ Chat, AOL IM, Yahoo Messenger, Chatrooms, Mirc.

Uno de los aspectos más interesantes en el estudio de herramientas de este estilo es la existencia o no de protocolos explícitos.

El modelo del chat es un modelo anárquico, donde cada usuario puede hacer lo que desee sin que la aplicación le imponga restricciones sobre las acciones permitidas. Por lo tanto se crea un protocolo tácito, donde los usuarios esperan a que su interlocutor escriba para luego responderle. Este protocolo es más difícil de respetar en chats grupales, donde sería necesario coordinar explícitamente la comunicación.

Otra posibilidad es tener un moderador, por ejemplo si deseamos modelar un debate entre políticos, donde uno de las dos personas tiene la palabra y luego de hablar, se calla y escucha al otro, o al moderador. La tarea del moderador es dirigir el debate hacia el lado que le interesa a la gente que esta escuchando o cerrar ideas de la exposición realizada previamente. Muchos modelos de comunicación tienen protocolos específicos (p.e.: Brainstorming, Debates, etc.) los cuales no son modelados ni soportados por las herramientas existentes.

Debe también notarse que el modelo actual de este tipo de herramientas se limita a considerar un aporte como caracteres individuales o a lo sumo líneas de texto marcadas probablemente con una indicación de autor y hora de creación. No se proveen, por lo general, mecanismos para aumentar los aportes combinando otros medios no textuales ni enriqueciendo su semántica.

La falta de riqueza y flexibilidad en estos aspectos trae como consecuencia, al menos, que se desconozca en gran medida la dirección que la comunicación por medio de estas herramientas debería tomar. Se ha llevado a cabo muy poca evaluación y experimentación respecto a las direcciones a seguir y su importancia.

Mi hipótesis es que esta falta de experiencia y evolución en esta área se debe a la carencia de herramientas innovadoras y flexibles que permitan recrear los distintos contextos

adecuados para cada tipo de comunicación. Si pretendemos avanzar en esta área necesitamos proveer mecanismos de construcción y adaptación de herramientas capaces de evolucionar con la velocidad y flexibilidad que el área requiere.

Contribuciones del trabajo de Tesis

El principal objetivo de esta tesis es el estudio de herramientas colaborativas, en particular, los chats. Estas herramientas de comunicación basadas en texto, no han evolucionado a la par de otros tipos de herramientas colaborativas. La forma de aportar soporte para la evolución e investigación de dichas herramientas es proveyendo u framework para la creación de herramientas chat con aspectos innovadores para este tipo de aplicaciones.

Como consecuencia de la necesidad de un ambiente de experimentación sobre aplicaciones concretas, surge la idea de implementar un prototipo del framework para la realización de herramientas chat, dicho framework lleva el nombre de Chatblocks.

También cabe destacar que este framework será utilizado por un organismo de investigación alemán, GMD, para la realización de determinadas herramientas chat como parte de un proyecto de comunicación a distancia llamado L3

El capitulo 2 estudia las herramientas chats dentro del contexto de aplicaciones gropuware, sus aspectos, similitudes y diferencias.

Los capítulos 3 y 4 proveen una introducción a las técnicas utilizadas para la realización del framework.

Los capítulos 5 y 6 expone la filosofía del framework, así como también interioriza al lector de la funcionalidad provista por el framework, sus componentes, formas de resolver determinadas situaciones, etc.

El capitulo 7 explica que componentes se utilizaron para la creación de cada herramienta del prototipo entregado junto a la documentación de la tesis.

Los capítulos 8 y 9 completan esta tesis agregando las conclusiones y la bibliografía de la misma.

Los apéndices A y B se componen de los diagramas de clase de Chatblocks y de las aplicaciones realizadas con dicho framework respectivamente.

Capítulo 2

Los chats en el contexto de aplicaciones colaborativas

En el área de comunicación mediante el uso de la computadora, encontramos varios tipos de herramientas, entre ellas, el Chat. Los chats generan un espacio común entre varios usuarios, donde cada uno puede comunicarse con los demás. La idea del chat, es que varias personas se junten en algún lugar, para discutir ideas, planificar, etc. El tipo y modalidad de la comunicación depende de la herramienta que estos utilizan, ya que en este tipo de aplicaciones podemos diferenciar varios aspectos que determinan su funcionalidad:

- Numero de Participantes
- Desarrollo de la conversación
- Unidad de Envío
- Almacenamiento
- Clasificación de Usuarios
- Floor Control
- Receptor
- Formas de recepción
- Formato de Envío
- Extensiones
- Semántica de los mensajes
- Criterio de Ordenación
- Visualización de los mensajes
- Awareness
- Representación del autor del mensaje
- Post-Edición
- Filtros

Cada uno de estos aspectos posee varias modalidades, las aplicaciones cubren el aspecto al proveer funcionalidad capaz de soportar una o más de sus modalidades. Cuanto más modalidades cubra, más flexible será la herramienta. Por ejemplo con el aspecto **Desarrollo de la conversación**, por lo general, las herramientas chat cubren una sola modalidad, es decir, son *sincrónicas* o *asincrónicas*. Una herramienta que cubra las dos modalidades, que pueda ser *sincrónica* y *asincrónica* a la vez, permitiendo que el usuario decida de qué forma desea desarrollar la conversación, será mucho más flexible que las anteriormente mencionadas.

Las modalidades de los aspectos se detallan a continuación:

Numero de Participantes: Cantidad de usuarios que participan de la conversación.

- *Dos*: Este tipo de aplicaciones centra su funcionalidad en permitir que dos personas se comuniquen de forma rápida (por lo genera es una comunicación sincrónica), el emisor del mensaje, focaliza el contenido en el receptor del mismo.
- *Mas de dos*: Estas herramientas permiten que varias personas a la vez, y en el mismo espacio virtual, se comuniquen entre sí. Aquí el numero de participantes puede llevar a confusiones y problemas que no se tienen en conversaciones cara a cara, ya que cuando el emisor de un mensaje quiere referirse a un receptor especifico, lo debería explicitar en el mensaje (en general, en una conversación real, el hecho de mirar a una persona mientras se esta hablando, implica que el receptor del mensaje es este).

En los chats de dos participantes los mensajes son más precisos y concretos, con mas de dos personas, si no existe un solo hilo de conversación, las subconversaciones pueden generar confusiones o interferencias entre sí.

Desarrollo de la conversación: Determina la forma de ejecución de la conversación.

- *Sincrónica*: Significa que todos los participantes deben estar conectados (ONLINE) para poder participar de la conversación, el hecho de desconectarse implica el abandono de la conversación.
- *Asincrónica*: No impone la necesidad de la conexión, de cada usuario, durante en transcurso de la conversación.

Los chats sincrónicos permiten la rápida interacción entre los participantes, pero exige la disponibilidad de estos en determinados momentos (todavía seria mas difícil hacer coincidir los horarios de dos personas que estén en diferentes países), ya que existen restricciones de tiempo, los mensajes son mas cortos y poco elaborados. Por otra parte los chats asincrónicos, permiten al usuario elaborar sus ideas, buscar información y completar los mensajes que envía a los demás participantes. No exige disponibilidad alguna sobre los tiempos del usuario, eso lleva a una disminución de la interacción y un desarrolla mas lento de las charlas, las cuales pueden durar días o meses.

Unidad de Envío: Indica cual es la unidad de envío a los demás participantes.

- *Carácter*: Cada carácter ingresado por el emisor en su aplicación local es enviado a los receptores.

- *Mensaje*: El emisor escribe el mensaje localmente y luego mediante una acción determinada lo envía a los receptores.

Los chats orientados a carácter proveen una rápida interacción entre los participantes, pero también permite que los receptores vean los errores del emisor y su frecuencia de tipeo, llevando a una disminución de aportes por parte de los usuarios inexpertos, bajando la interacción de la conversación.

Almacenamiento: Indica cuan accesible es la información de una conversación.

- *Volátil*: Los mensajes son almacenados temporalmente, mientras la conversación transcurre, y cuando la aplicación se cierra, se eliminan.
- *Permanente*: Todos los mensajes son accesibles durante y después que la conversación transcurra.

Algunos chats de almacenamiento volátil, proveen la funcionalidad de guardar la conversación, que actualmente esta almacenada temporalmente, en archivos con formatos accesibles por otras aplicaciones. : Html, Doc, Txt, etc.

Clasificación de Usuarios: Se referencia a la diferenciación de los usuarios, desde el punto de vista de la aplicación, esto influye directamente en la funcionalidad de la aplicación. Para cada rol de usuario, la aplicación debe proveer diferente funcionalidad.

- *Sin roles*: Todos los usuarios son iguales.
- *Con roles*: Existe una diferenciación entre los usuarios. Entre los roles implementados en chats convencionales, encontramos:
  - **Moderador**: Es el administrador de la conversación, es el encargado de dirigir la conversación.
  - **Usuario**: Este es un participante común de la conversación, por lo general, solo se le permite aportar sentencias a la conversación.

La existencia de roles es un aspecto no explotado en los chats comerciales, ya que en la mayoría solo existen usuarios comunes. Por ejemplo, el concepto que un usuario sea el moderador de la conversación, permitiendo a este dirigir la misma, solo esta implementado en algunos chats asincrónicos.

Floor Control El control de piso es un aspecto muy utilizado en groupware, indica, si existe alguna restricción, quien es el usuario habilitado a desarrollar una acción determinada, por ejemplo, en un juego de ajedrez, indica cual es el jugador que puede realizar la movida, en el caso particular de los chats, realizar un aporte (enviar un mensaje). Esta directamente relacionado con el uso de roles en la conversación. Este indica a la aplicación como debe responder a ciertas situaciones dependiendo de los roles del usuario local y los remotos.

- *Ninguno*: No existe un protocolo establecido. Que la aplicación no exija cumplir con un protocolo no significa que la conversación no sigue un protocolo, como en toda reunión humana, este existe y es implícito.
- *Protocolo*: La aplicación reconoce situaciones y fuerza al usuario a realizar cierto tipo de acción, por ejemplo, esperar hasta que otro participante hable. El uso de roles facilita y generaliza esta tarea, ya que las restricciones se definen sobre un rol y no sobre los usuarios.

La mayoría de los chats comerciales no cubren este aspecto, lo que fuerza a los usuarios a cumplir con un protocolo, tácito, por ejemplo, luego de hablar, esperar para leer que tiene el otro participante para decir, aunque la aplicación le permita volver a hablar. El uso de un protocolo explícito, es un punto importante en la experimentación de nuevas herramientas, ya que posibilitan reproducir situaciones reales, como por ejemplo, un debate entre dos personas o conferencia de prensa.

Receptor: Indica quien recibirá un mensaje emitido por un participante.

- *Todos*: Cada mensaje es replicado a todos los participantes de la conversación.
- *Algunos*: El emisor puede establecer una propiedad en el mensaje que indica que solo lo deberán leer determinados usuarios.

La función que permite definir los receptores de un mensaje se llama, en la mayoría de los chats, "Private".

Formas de recepción: Indica quien recibirá un mensaje emitido por un participante.

- *Solo los emitidos mientras el receptor esta ONLINE*: El receptor solo recibe mensajes cuando esta online, todos los mensajes emitidos mientras este se encuentra offline, se pierden.
- *Todos*: Al receptor llegan todos los mensajes emitidos.

Algunos chats sincrónicos poseen un mecanismo de buffering de mensajes y cuando un usuario ingresa al sistema, este le envía todos los mensajes que tiene guardados.

Formato de Envío: Establece el formato de la información que se enviará.

- *Texto Plano*: Secuencia de caracteres.
- *Texto Formateado*: Texto, con la posibilidad de definir tipo de letra, tamaño, color y estilos de los caracteres.
- *Imágenes*: Permite incluir imágenes como iconos, o bitmaps.

Las herramientas convencionales solo permiten el envío de texto (con formato o sin el) y en algunas, también es posible agregar iconos o pequeñas imágenes, al texto que enviamos.

El poder combinar estos tipos de formatos, otorga al usuario una mayor capacidad expresiva a la hora de plasmar una idea.

Extensiones: Existen aplicaciones que sirven de apoyo a al conversación, estas permiten el envío de nuevos formatos de información, permitiendo enriquecer mas aun la comunicación.

- *Audio*: Similar a las comunicaciones telefónicas.
- *Video*: El emisor envía una secuencia de imágenes, captadas por una cámara a una frecuencia determinada, que al reproducirlas en orden provocan el efecto de movimiento continuo.
- *Pizarra*: Se pueden realizar dibujos compartidos, cada usuario puede ir agregando su aporte para la construcción de un solo dibujo, el cual es visto por todos los usuarios.

Cada uno de estos tipos de aplicaciones son una rama dentro de las comunicaciones mediante la computadora, por eso no voy a introducirme mucho en sus respectivos dominios.

El hecho a destacar es que hoy en día se deben recurrir a distintas aplicaciones para poder realizar una conversación productiva, combinándolas para poder cubrir todos los tipos de comunicación conocidos, lo cual denota una falta de completitud en cada una de estas aplicaciones.

Las herramientas de chat mas modernas poseen extensiones de este tipo, esto se debe a una tendencia en los usuarios mas expertos, quienes a la hora de comunicarse utilizan varias aplicaciones a la vez, explotando sus ventajas y supliendo sus falencias con las ventajas de otra herramienta de comunicación. Como ejemplo podemos citar, el uso de un chatroom combinado con la utilización de audio y video.

Semántica de los mensajes: Expresa el significado de un mensaje desde el punto de vista de la aplicación.

- *Ninguna*: No existe significado alguno, mas allá de su contenido, los mensajes son solo eso mensajes.
- *Respuesta*: Es un mensaje que referencia a uno anterior, continua el hilo de conversación (thread).
- *Referencia a un objeto*: Se refiere a un objeto en particular, este puede ser interno o externo con respecto a la aplicación.

La semántica es un aspecto poco explotado en los chat convencionales, y los pocos que la utilizan en situaciones aisladas.

Criterio de Ordenación: Indica el orden en el cual se verán los mensajes.

- *Atributos*: La ordenación esta basada en atributos propios del mensaje, como por ejemplo la fecha de creación o su autor.
- *Semántica*: Utiliza el significado del mensaje para la ordenación, esta puede ser por ejemplo, el hilo de la conversación (thread) ordenando los mensajes e inmediatamente después sus respuestas.

La ordenación de los mensajes es un aspecto poco explotado, por lo general estos son ordenados por la fecha de su creación, pero a veces este tipo de ordenación puede llevar a confusiones por parte de los participantes de la conversación, teniendo que reenviar mensajes para desambiguar ciertas situaciones.

Visualización de los mensajes: Este aspecto, hace referencia a como se verán los mensajes, tiene mucha relación con la ordenación de los mismos.

- *Lista*: Los mensajes se muestran uno detrás del otro.
- *Arbol*: Los mensajes forman una jerarquía, y estos se muestran en forma de árbol.

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.

Awareness: Es la información, que un participante necesita enviar a los demás, para simular la presencia en el ambiente virtual creado por las diferentes aplicaciones cooperativas, en este caso la conversación.

- *De Participación*: Se conocen quienes participan de la conversación.
- *De Conexión:* De cada participante, se conoce si esta conectado a la aplicación (ONLINE) o no.
- *De Tipeo*: De cada participante, se conoce si esta escribiendo, es decir esta preparando un mensaje para ser enviado.

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 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 participación, ya que adelanta la participación de un usuario en la conversación.

Representación del autor del mensaje: Consiste en otorgar la identidad del autor a cada elemento, por ejemplo, en el caso de un mensaje, que se pueda identificar su autor, sin requerir mas información que el icono del usuario.

- *Nombre*: Se utiliza el nombre de usuario, o su login a la aplicación, para representarlo.

- *Tipo y Color de Font*: Cada usuario escoge un tipo y color de font para la representación de los mensajes de su autoria, cuando al receptor le llega un mensaje, este se muestra con tipo y color de font del autor.
- *Imágenes*: Cada usuario es referenciado por una imagen, cuando un mensaje es emitido, la imagen asociada de su autor es enviada junto con el mensaje.

En el caso de las imágenes o tipos de fonts, hay que tener en cuenta que estas se pueden repetir, cuando la cantidad de participantes excede cierto numero. También, la aplicación, deberia ofrecer la opción de ver las referencias a la relación entre usuario y su representación, la cual es de gran utilidad durante los primeros mensajes del usuario, hasta que todos los demás participantes incorporen la representación del nuevo participante.

Post-Edición: Especifica las operaciones, cuya funcionalidad consiste en corregir los contenidos de la conversación.

- *No Permite*: Una vez enviado un mensaje, esto no puede ser corregido en ningún momento.
- *Borrado*: Un mensaje, previamente enviado, puede eliminarse.
- *Edición*: Se pueden corregir los contenidos o referencias / relaciones de los mensajes posteriormente al envío de los mismos.

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.

Filtros: Expresa como se puede filtrar la información de la conversación, en este caso los mensajes.

- *Emisor*: Los mensajes de determinado autor, o emisor, son.
- *Contenido*: Se toman en cuenta solo los mensajes con cierto contenido.
- *Semántica*: En esta caso, los mensajes deben poseer semántica, y sobre se pueden realizar varios filtros. Por ejemplo todos los mensajes de un thread determinado, todas las referencias a determinados objetos, etc.

El filtrado de mensajes es un aspecto que proveen varias aplicaciones.

Clasificación de herramientas chat.

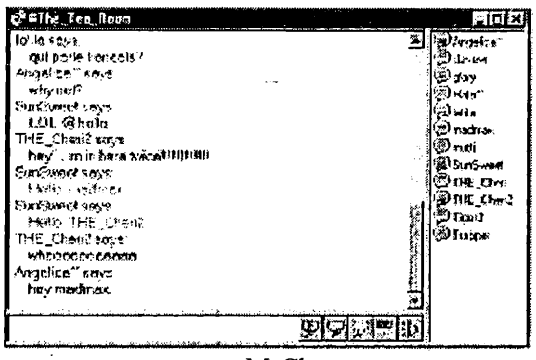
En esta sección se analizarán cuatro tipo de herramientas. Las mismas conforman el 90% de las herramientas utilizadas para conversaciones mediante computadora.

Chatroom

Este tipo de herramientas se utiliza, generalmente, como plugin de una pagina web, a la cual el usuario se registra y luego comienza a chatear. Utilizado para emular un cuarto donde se encuentran varias personas a la vez. Un usuario entra al cuarto (chatroom) y se encuentra con todos los que están dentro de este. Como aplicación comercial de este tipo, encontramos **msChat** y otras de similares características pero de menor renombre.

Las características de este tipo de aplicación son las siguientes:

- **Filtros:** No Permite.
- **Unidad de Envío:** Mensaje.
- **Almacenamiento:** Volátil.
- **Visualización:** Lista.
- **Semántica de los Mensajes:** Ninguna.
- **Avatars:** Nombre, Color de Fuente.
- **Receptor:** Todos los que están en el mismo chatroom.
- **Nro de Participantes:** 2 o más.
- **Formas de Recepción:** Solo estando Online.
- **Formato de Envío:** Texto Plano.
- **Desarrollo de la conversación:** Sincrónica.
- **Ordenación:** Fecha de emisión.
- **Awareness:** Conexión.
- **Floor Control:** Ninguno.
- **Clasificación de Usuarios:** Sin Roles
- **Corrección de contenidos:** No Permite.



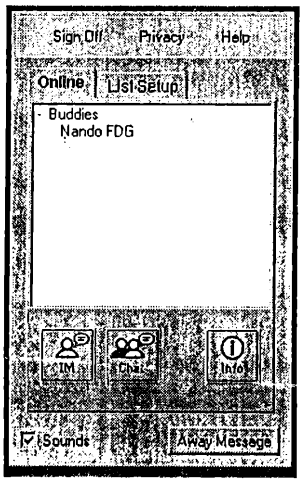
MsChat

Instant Messenger

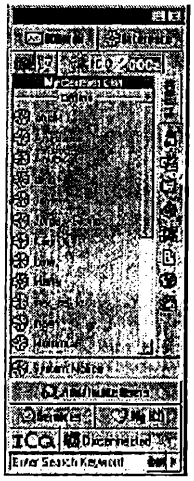
En este caso, la herramienta es aplicación local, la cual registra los datos del usuario y al ejecutar se conecta a un servidor, que es el que supervisa todas las actividades de los usuarios. Cada usuario tiene una lista de contactos, las personas con las que desea conversar. Cuando un usuario se conecta, todos los que lo tengan en su lista de contactos se enterrarán que él se conectó. Dentro de este tipo de aplicaciones, las mas utilizadas son **AOL IM**, **Yahoo Messenger** e **ICQ**.

Las características de este tipo de aplicación son las siguientes:

- **Filtros:** No Permite.
- **Unidad de Envío:** Mensaje.
- **Almacenamiento:** Permanente (en disco local).
- **Visualización:** Lista.
- **Semántica de los Mensajes:** Ninguna.
- **Avatars:** Nombre, Color de fuente.
- **Receptor:** Algunos.
- **Nro de Participantes:** 2.
- **Formas de Recepción:** Solo estando Online.
- **Formato de Envío:** Texto con formato.
- **Desarrollo de la conversación:** Sincrónica.
- **Ordenación:** Fecha de emisión.
- **Awareness:** Conexión.
- **Floor Control:** Ninguno.
- **Clasificación de Usuarios:** Sin Roles
- **Corrección de contenidos:** No Permite.



AOL IM



ICQ

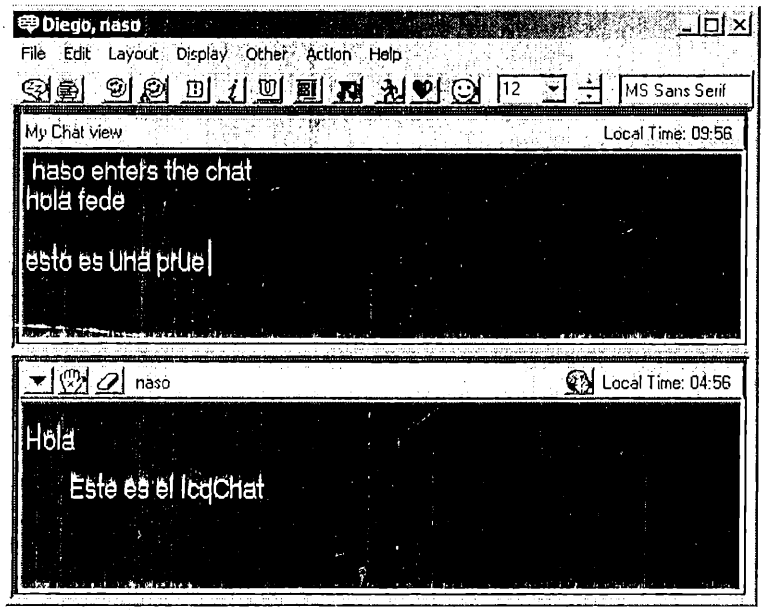
Chat Sincrónico

Los primero chats desarrollados son de este estilo, estos cuentan con una ventana de texto dividida en dos secciones, donde cada usuario escribe en una de ellas. Ambos pueden ver que esta escribiendo el otro.

. En cuanto a aplicaciones este tipo encontramos: Talk (S.O. UNIX) e **ICQChat**, la misma es una subaplicación del ICQ.

Las características de este tipo de aplicación son las siguientes:

- **Filtros:** No Permite.
- **Unidad de Envío:** Caracteres.
- **Almacenamiento:** Volátil.
- **Visualización:** Lista.
- **Semántica de los Mensajes:** Ninguna.
- **Avatars:** Tipo de Font.
- **Receptor:** Uno solo, el otro participante de la conversación.
- **Nro de Participantes:** 2.
- **Formas de Recepción:** Solo estando Online.
- **Formato de Envío:** Texto Plano.
- **Desarrollo de la conversación:** Sincrónica.
- **Ordenación:** Fecha de emisión.
- **Awareness:** Tipeo.
- **Floor Control:** Ninguno.
- **Clasificación de Usuarios:** Sin Roles
- **Corrección de contenidos:** Borrado.

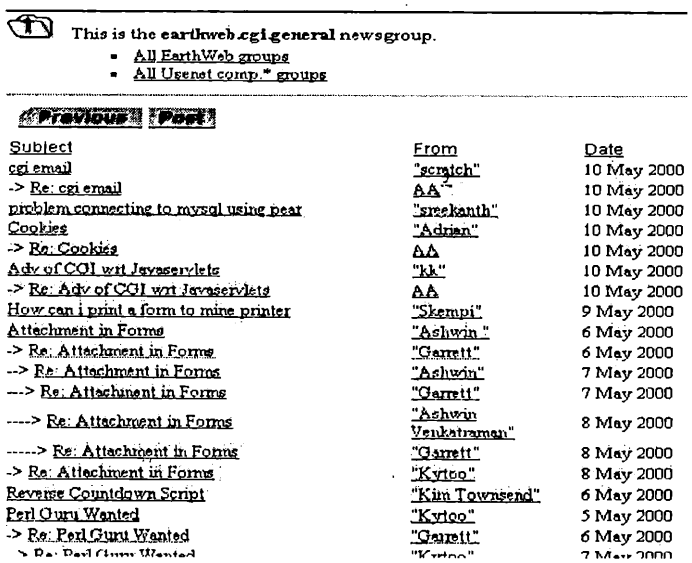


Chat Asincrónico

Este tipo de chat son desarrolladas, por lo general, como aplicaciones web. Las mismas se presentan al usuario como una pagina HTML, el usuario accede a los diferentes mensajes por medio de links. También reciben el nombre de forum.

Las características de este tipo de aplicación son las siguientes:

- **Filtros:** Emisor, Semántica.
- **Unidad de Envío:** Mensaje.
- **Almacenamiento:** Permanente.
- **Visualización:** Arbol.
- **Semántica de los Mensajes:** Respuesta, continúa un Thread.
- **Avatars:** Nombre.
- **Receptor:** Todos.
- **Nro de Participantes:** 2 o más.
- **Formas de Recepción:** Todos los mensajes enviados.
- **Formato de Envío:** Texto Formateado.
- **Desarrollo de la conversación:** Asincrónica.
- **Ordenación:** Fecha de emisión, Thread, Emisor.
- **Awareness:** Participación.
- **Floor Control:** Ninguno.
- **Clasificación de Usuarios:** Sin Roles
- **Corrección de contenidos:** No Permite.



This is the **earthweb.cgi.general** newsgroup.

- All EarthWeb groups
- All Usenet comp.* groups

Previous Post

Subject	From	Date
cgi email	"scratch"	10 May 2000
-> Re: cgi email	AA	10 May 2000
problem connecting to mysql using pear	"sreekanth"	10 May 2000
Cookies	"Adrian"	10 May 2000
-> Re: Cookies	AA	10 May 2000
Adv of CGI wrt Javaservlets	"kk"	10 May 2000
-> Re: Adv of CGI wrt Javaservlets	AA	10 May 2000
How can i print a form to mine printer	"Skempi"	9 May 2000
Attachment in Forms	"Ashwin "	6 May 2000
-> Re: Attachment in Forms	"Garrett"	6 May 2000
--> Re: Attachment in Forms	"Ashwin"	7 May 2000
---> Re: Attachment in Forms	"Garrett"	7 May 2000
----> Re: Attachment in Forms	"Ashwin Venkatraman"	8 May 2000
-----> Re: Attachment in Forms	"Garrett"	8 May 2000
-> Re: Attachment in Forms	"Kyrtoo"	8 May 2000
Reverse Countdown Script	"Kim Townsend"	6 May 2000
Perl Guru Wanted	"Kyrtoo"	5 May 2000
-> Re: Perl Guru Wanted	"Garrett"	6 May 2000

Chat Asincrónico

Capítulo 3

Frameworks de Aplicación Orientados a Objetos

El poder de cómputo y el ancho de banda de las redes han aumentado dramáticamente en la última década, todavía el diseño e implementación de software complejo permanece caro y propenso a errores. Mucho del costo y esfuerzo proviene del continuo redescubrimiento y reinvención de conceptos centrales y componentes, por la industria del software. En particular, la heterogeneidad creciente de arquitecturas de hardware y la diversidad de sistemas operativos y plataformas de comunicación hace difícil construir aplicaciones correctas, portátiles, eficaces, y baratas desde el principio.

Los frameworks de aplicación orientados a objetos son una tecnología prometedora para la transformación de aplicaciones, tanto para su diseño como para su implementación, para reducir el costo y mejora la calidad del software. Un framework es una aplicación semicompleta reusable, que puede ser especializado para producir aplicaciones personalizadas. En contraste al antiguo reuso orientado a objetos, técnicas basadas en librerías de clase, los frameworks apuntan a un mercado en particular (como procesamiento de datos o en comunicaciones celulares) y aplicaciones de dominio (como interfaces de usuario). Frameworks como MacApp, ET++, Interviews, ACE, el MFC y DCOM de Microsoft, el RMI de JavaSoft, e implementaciones de CORBA de la OMG juegan un papel importante en el desarrollo de software contemporáneo.

Los beneficios primarios de un framework de aplicación orientado a objeto provienen de la modularidad, reusabilidad, extensibilidad, e inversión de control que ellos proporcionan desarrolladores.

Los frameworks refuerzan la modularidad encapsulando los detalles de implementación volátil detrás de las interfaces estables. La modularidad de los frameworks ayudan a mejorar la calidad del software, localizando el impacto en los cambios de diseño e implementación, que reducen el esfuerzo requerido para entender y mantener el software existente.

Las interfaces estables, proporcionadas por los frameworks, refuerzan la reusabilidad definiendo componentes genéricos que se pueden volver a aplicar para crear nuevas aplicaciones. El conocimiento del dominio y el esfuerzo anterior de desarrolladores experimentados, influyen en la reusabilidad del framework, para evitar recrear y revalidar soluciones comunes a los requisitos de la aplicación y desafíos del diseño del software. La reutilización de los componentes del framework conduce a mejorar substancialmente la productividad del programador, así como elevar la calidad, el funcionamiento, la confiabilidad y la interoperabilidad del software.

Un framework mejora la extensionalidad proporcionando enganche explícito de métodos, esto permite que las aplicaciones extiendan sus interfaces estables. El enganche sistemático de los métodos separa las interfaces estables y el comportamiento de una aplicación de las variaciones requeridas por las instancias de una aplicación en un contexto particular. La extensionalidad de los frameworks es esencial para asegurar futuras adaptaciones, como nuevos servicios o aspectos de las aplicaciones. La característica principal de las arquitecturas de los frameworks es la inversión

de control. Esta arquitectura establece los pasos a seguir por la aplicación, para luego ser personalizados por handler del objeto que invoca el evento por medio del mecanismo de dispatching del framework. Cuando los eventos ocurren, el dispatcher del framework reacciona invocando los métodos de enganche prerregistrados por el handler del objeto que realiza evento. La inversión de control permite al framework (en lugar de cada aplicación) determinar cuáles, de los métodos específicos de la aplicación, invocar en contestación al evento externo (como mensajes que llegan de los usuarios o paquetes que arriban de un puerto de comunicación en particular).

Frameworks utilizados masivamente

Los desarrolladores, en ciertos dominios, han tenido éxito al utilizar Frameworks de Aplicación Orientados a Objetos para el desarrollo de aplicaciones durante muchos años. Los primeros frameworks orientados a objetos (como MacApp e Interviews) se originaron en el dominio de interfaces gráficas de usuario. El Microsoft Fundation Clases (MFC) es un framework contemporáneo de GUI que se ha vuelto el estándar de facto de la industria para crear aplicaciones gráficas en plataformas de PC. Aunque MFC tiene limitaciones (como la falta de portabilidad a plataformas no-PC), su adopción extendida demuestra los beneficios en la productividad, de reusar frameworks para desarrollar aplicaciones gráficas comerciales. Desarrolladores de aplicación en dominios más complejos (como telecomunicaciones o aeroelectrónica de tiempo real) tienen tradicionalmente una falta de frameworks estándar. Como resultado, los diseñadores en estos dominios, crean, validan y mantienen sistemas de software desde el principio. En la era de la desregulación y la competencia global, el desarrollo de aplicaciones completamente in-house, se ha vuelto sumamente caro, en tiempo y dinero.

Afortunadamente, la próxima generación de frameworks de aplicación Orientados a Objetos esta apuntando a negocios y dominios de aplicaciones complejas. En el centro de estos esfuerzos están los Object Request Broker frameworks (ORB) que facilitan la comunicación entre objetos locales y remotos. Los frameworks ORB eliminan muchos aspectos, tediosos, propensos a errores, y no portátiles, de la creación y mantenimiento de aplicaciones distribuidas y componentes de servicio reusables. Esto permite que los programadores diseñen y desarrollen aplicaciones complejas rápida y robustamente, en lugar de estar luchando eternamente con problemas de bajo nivel de la infraestructura.

Clasificación de los Frameworks de Aplicación

Aunque los beneficios y principios de diseño subyacentes de los frameworks son en gran medida independientes de los dominios a los que ellos son aplicados, existe varias clasificaciones de frameworks

Clasificación de Frameworks por su alcance:

- *System infraestructure frameworks*: Simplifican el desarrollo de sistemas de infraestructura portables y eficientes, como sistemas operativos, frameworks de comunicaciones y frameworks para las interfaces del usuario. Los System infraestructure frameworks se usan, principalmente, dentro de una organización o empresa de software y no se vende a clientes directamente.

- *Middleware integration frameworks:* Se usan normalmente para integrar aplicaciones distribuidas y componentes. Los Middleware integration frameworks están diseñados para reforzar la habilidad, de los desarrolladores de software, de modularizar, reusar, y extender su infraestructura de software, ocultando el ambiente de trabajo distribuido al programador. Los Middleware integration frameworks representan un mercado en crecimiento. Los ejemplos comunes incluyen ORB frameworks, middleware orientado a los mensajes, y bases de datos transaccionales.

- *Enterprise application frameworks:* Están dirigidos a dominios de aplicación (como telecomunicaciones, aeroelectrónica, fabricación e ingeniería financiera, es la piedra angular de actividades comerciales de la empresa. Los enterprise frameworks son mas caros de implementar y/o comprar, con respecto a los Middleware integration frameworks y System infraestructure Frameworks. Sin embargo, los Enterprise application frameworks proveen un retorno sustancial en inversión, ya que estos soportan directamente el desarrollo de aplicaciones de usuario final y productos. En contraste, los Middleware integration frameworks y System infraestructure Frameworks se enfocan en problemas de desarrollo de software interno.

Aunque estos frameworks son esenciales para crear software de alta calidad rápidamente, ellos, por lo general, no generan rédito sustancial para las grandes empresas. Como resultado, es a menudo más rentable comprar Middleware integration frameworks y System infraestructure Frameworks en lugar de desarrollarlo.

Clasificación de Frameworks por su técnica de extensión:

- *White Box Frameworks*: Confían en lenguajes orientados a objetos rasgos como la herencia y el binding dinámico para adquirir extensionalidad. La funcionalidad existente se reusa y se extiende por:
  1. Heredando de la clases base del frameworks.
  2. Sobreescribiendo métodos de enganche predefinidos usando patterns como el Template Method.

- *Black Box Frameworks:* Soportan extensionalidad definiendo interfaces para componentes, los mismos pueden conectarse al framework vía composición de objetos. La funcionalidad existente se reusa y se extiende por:
  1. Definiendo componentes que implementen una interface particular.
  2. Integrando estos componentes al framework, usando patters como Strategy y Functor.

Los frameworks white-box requieren que los desarrolladores de la aplicación posean un conocimiento íntimo de la estructura interior del framework.. Aunque los frameworks white-box son ampliamente usados, ellos tienden a producir sistemas que se acoplan herméticamente a los detalles específicos de la jerarquía de herencia del framework. En contraste, los frameworks black-box se estructuran a usando composición y delegación en lugar de herencia.

Como resultado, los frameworks black-box son generalmente más fáciles de usar y extender que frameworks white-box. Sin embargo, los frameworks black-box son más difícil desarrollar, ya que ellos requieren que los diseñadores de framework definan interfaces y enganches que se anticipen a los casos de uso futuros.

Ventajas y desventajas de los frameworks de aplicación

Cuando se usan junto con patterns, librerías de clase, y componentes, los frameworks orientados a objetos elevan significantemente la calidad del software y reducen en esfuerzo de desarrollo. Sin embargo, se deben realizar varios desafíos para emplear efectivamente los frameworks. Compañías que intentan construir o usar frameworks demasiado reusables fallan a menudo, a menos que ellos reconozca y resuelvan desafíos como esfuerzo de desarrollo, learning curve, integrabilidad, mantenibilidad, validación y debugging, eficiencia y falta de standards.

- El desarrollo de software complejo es bastante dificultoso, el desarrollo de frameworks de alta calidad, extensibles, y reusables para aplicaciones de dominios complejos son aun más complicados. A menudo, las habilidades requeridas para producir frameworks exitosos, permanecen retenidas en las cabezas de diseñadores especialistas.

- Aprender a usar un framewor de aplicación orientado a objetos eficazmente requiere una inversión considerable de esfuerzo. Por ejemplo, toma a menudo de 6 a 12 meses para ser muy productivo con un framework de GUI como MFC o MacApp, dependiendo de la experiencia de los desarrolladores. Generalmente, como parte del entrenamiento, se requieren cursos que le enseñen a los desarrolladores, cómo usar el framework eficazmente. A menos que el esfuerzo requerido para aprender un framework pueda ser amortizado en muchos proyectos, esta inversión no es efectiva con respecto a su costo. Es más, la conveniencia de un framework para una aplicación particular, no esta definida hasta que la curva de aprendizaje se haya nivelado.

- El desarrollo de aplicaciones tiende a convertirse cada vez más en la integración de múltiples frameworks (Ej. GUIs, sistemas de comunicación, los bases de datos) junto con las librerías de clase, sistemas de legado y componentes existentes. Sin embargo, muchos de los frameworks de ultima generación fueron diseñados para extensión interna, en lugar de la integración con otros frameworks desarrollados externamente. Los problemas de integración se presentan en varios niveles de abstracción, yendo des los problemas de la documentación, arquitectura de concurrencia / distribución, hasta event despatching del modelo.

- Los requisitos de las aplicación cambian frecuentemente. Por consiguiente, en normal que los requisitos del framework a menudo también cambian. Cuando los frameworks evolucionan, las aplicaciones que lo usan, deben desarrollarse con ellos.

- Las actividades de mantenimiento de frameworks incluyen modificación y adaptación del framework. Ambos la modificación y la adaptación pueden ocurrir a un nivel funcional (es decir, cierta funcionalidad del framework no cubre totalmente los requerimientos de los diseñadores), así como en un nivel no funcional (qué incluye aspectos más cualitativos como portabilidad o reusabilidad).

- El mantenimiento del framework puede tomar diferentes formas, como agregar funcionalidad, eliminar funcionalidad, y generalización. Un profundo entendimiento de los componentes del framework y sus relaciones mutuas son esenciales realizar esta tarea con éxito. En algunos casos, el desarrollador de la aplicación y/o los usuarios finales deben confiar completamente diseñadores del framework para mantener el framework.

- Aunque un framework modular bien diseñado puede localizar el impacto de defectos del software, validación y puesta a punto aplicaciones construidas usando framework, esto puede ser engañoso por las razones siguientes:

    3. Los componentes genéricos son más difíciles de validar en lo abstracto. Un componente del framework bien diseñado, generalmente evita detalles específicos de la aplicación, estos son provistos vía subclasificación, composición de objetos o parametrización de templates. Mientras que esto mejora la flexibilidad y extensibilidad del framework, complica el testeo de módulos, ya que los componentes no pueden validarse estando aislados de su instancias específicas. Generalmente es difícil distinguir si un error es producido por el framework o el código de la aplicación. Como en cualquier desarrollo de software, los errores en un framework pueden

ser introducidos de muchas formas posibles, como falta de entendimiento de los requisitos, diseño demasiado acoplado, o una incorrecta implementación. Al personalizar los componentes en un framework a una aplicación particular, el numero de posibles fuentes de error aumentará.

4. Inversión de control y falta de flujo de control explícito. Aplicaciones escritas con frameworks son mas difíciles de poner a punto, ya que el fluyo de control "invertido" del framework oscila entre la infraestructura de la aplicación independiente del framework y los métodos específicos de la aplicación invocados. Esto aumenta la dificultad del "single-stepping" a través del comportamiento en tiempo de ejecución del framework dentro del debugger, ya que el flujo de control de la aplicación se maneja implícitamente por callbacks y los desarrolladores no pueden entender o no tienen acceso al código del framework.

Los frameworks refuerzan la extensibilidad empleando niveles adicionales de indirección. Por ejemplo, el binding dinámico es usado comunmente para permitir a los diseñadores subclasificar y personalizar interfaces existentes. Sin embargo, la generalización y flexibilidad a menudo reducen la eficiencia. Por ejemplo, en idiomas como C++ y Java, el uso de binding dinámico hacen impráctico el soporte de Concrete Data Types (CDTs), qué es a menudo requerido para software de tiempo real. La falta de CDTs conduce a:

1. Aumento del almacenamiento (debido a los punteros contenidos en tablas virtuales).

2. Degradación de performance (debido a la sobrecarga de invocar un método dinámicamente y la incapacidad mantener en línea métodos pequeños).

3. Falta de flexibilidad (debido a la incapacidad de almacenar objetos en memoria compartida).

Actualmente no hay ningún standard ampliamente aceptada, para diseño, implementación, documentación, y adaptación de frameworks. Es más, la industria emergente de frameworks standard (como CORBA, DCOM, y Java RMI) falla en la semántica, aspectos e interoperabilidad para ser verdaderamente eficaces por en los múltiples dominios de aplicación.

Consideraciones generales

Los frameworks de aplicación orientados a objetos serán el centro de la tecnología de software del siglo XXI. El enfoque extenso en frameworks de aplicación es, en la comunidad de diseñadores de software orientado a objetos, un vehículo importante para el reuso y un medio

para capturar la esencia de modelos, arquitecturas, componentes y mecanismos de programación exitosos.

Es significante que los frameworks están volviéndose la tendencia principal y que los diseñadores de todos los niveles están adoptando cada vez más los frameworks y teniendo éxito con sus tecnologías. Sin embargo, los frameworks de aplicación orientados a objetos son sólo tan buenos como las personas que los crea y usa. La creación de frameworks de aplicación robustos, eficaces y reusables requieren grupos de desarrolladores con una amplia gama de habilidades. Se necesitan analistas y diseñadores especialistas expertos en patterns, arquitecturas de software y protocolos para aliviar las complejidades inherentes y accidentales del software complejo. También es necesario desarrolladores especialistas en middleware que pueden implementar estos modelos, arquitecturas y protocolos en los frameworks reusables. Por ultimo, los programadores de la aplicación son quienes tienen la motivación, habilidades y entrenamiento para aprender a usar estos frameworks eficazmente.

Capítulo 4

COAST

Las aplicaciones cooperativas sincrónicas (groupware sincrónico) permiten a un grupo de usuarios trabajar conjuntamente y simultáneamente en datos compartidos, como un documento. Un ejemplo podría ser, un sistema de reserva de vuelos, donde las acciones de cada usuario serán visibles al instante para cada uno de los demás usuarios de la aplicación. La aplicación que se está consideración aquí requiere un alto grado de sincronización, es decir, las acciones realizadas por un usuario deben ser percibidas inmediatamente por los demás usuarios cooperativos. No es suficiente observar los cambios realizados sobre la fuente de datos en común, también se necesita información sobre el cambio del contexto, considerando quién comenzó la acción, y si posible, por qué fue comenzada. Recíprocamente, los usuarios deben ser conscientes que sus propias acciones pueden tener efectos en otros usuarios del sistema. Esta percepción de qué están haciendo otros miembros del grupo es conocida como *group awareness* (conocimiento de grupo), y sirve para ayudar a que los usuarios coordinen sus acciones dentro del grupo. El framework COAST pretende hacer el desarrollo de groupware sincrónico más fácil, con un costo comparable al desarrollo de aplicaciones monousuario equivalentes.

Paradigma de programación

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.

COAST ayuda a los diseñadores en tiempo de implementación a través de los siguientes aspectos:

- Diseño del modelo de dominio compartido usando el lenguaje de descripción de datos de COAST.
- Un modelo extensible predefinido de usuarios y sus ambientes de trabajo.
- Constructor de interfaces de usuario que proporciona widgets cooperativos.

En tiempo de ejecución las aplicaciones COAST obtienen los siguientes beneficios:

- La replicación de objetos es transparente.
- Procesamiento de transacciones para asegurar la consistencia de los datos.
- Actualización automática de views cuando un objeto compartido cambia.

Se han identificado tres del problema principales en el área:

- Consistencia de los documentos compartidos.
- La creación de una vista idéntica sobre estos documentos para todos los usuarios.
- El modelado de varias formas de cooperación y situaciones cooperativas.

Como respuesta a estos problemas, el GMD-IPSI construyó el framework COAST. Pueden crearse aplicaciones cooperativas basándose en este framework sin exceder el esfuerzo de desarrollo que se requeriría para una aplicación monousuario comparable.

Básicamente, esto se logra:

- Ocultando, 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.
- Ocultando, 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.
- Ocultando 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.

COAST asume que una aplicación consiste en objetos GUI y objetos documento.

Paradigma Model View Controller (MVC)

Los objetos GUI pueden ser considerados como *views* y *controllers*, los objetos documento pueden ser considerados como modelos. Los objetos GUI son responsables de presentar un documento en forma de gráfica e interpretar el ingreso de datos del usuario.

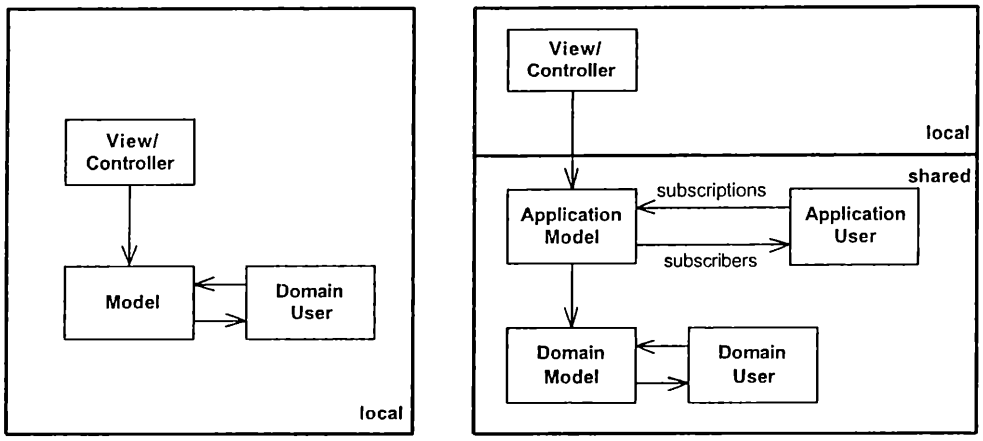
Los objetos documento son responsables de representar la información esencial del documento. En otras palabras, ellos representan la información que se guardaría en aplicaciones monousuario, esta generalmente es un archivo.

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

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 *group awareness* o *floor control*.

En el diseño de aplicaciones monousuario, la separación del modelo de la aplicación y modelo del dominio, ha demostrado resultados exitosos. Esta separación también es útil para el desarrollo de aplicaciones multiusuario, solo si están basadas en un espacio de objetos compartido y adaptada a las necesidades específicas del caso multiusuario. COAST transfiere este concepto a las aplicaciones colaborativas.



Modelo convencional (MVC) Modelo COAST

Comparación entre el modelo convencional, para aplicaciones monousuario, y el modelo COAST, para aplicaciones groupware.

Implementando objetos compartidos

El programador de la aplicación se libra de cualquier tarea relacionada con la replicación, manejo o transporte de los objetos compartidos. Para facilitar un buen control de concurrencia y manejo de replicaciones de los objetos compartidos sin involucrar al programador de la aplicación, COAST divide los objetos compartidos en dos categorías:

Acciones permitidas al programador de la aplicación.	Categoría 1: Objetos definidos por el programador de la aplicación.	Categoría 2: Objetos predefinidos del framework.
Instanciación	Sí	Sí
Métodos propios	Sí	No
Subclasificación	Sí	No

Todos los objetos compartidos, definidas por el programador, deben ser instancias (directamente o indirectamente) de la clase *CoastModel*, definida por el framework. A pesar de esta restricción, los objetos compartidos definidos por el programador pueden tener atributos y comportamiento propios.

Existen varias clases predefinidas por el framework. Al programador de la aplicación no se le permite subclasificarlas o cambiar su código. La ventaja de los objetos predefinidos por el framework y la razón por la cual el programador debe hacer uso de ellos es que, el framework puede proporcionar un eficaz control de concurrencia y manejo de replicación para estos objetos.

Un aspecto central de COAST, es el hecho que el programador de la aplicación hace uso de un paradigma de *frame-slot* cuando diseña los objetos documento. Los atributos de los objetos compartidos son considerados como *slots*. Comparado con los atributos convencionales, el uso de *slots* asegura, al programador de la aplicación, que su código produce estructuras consistentes. Al mismo tiempo, el sistema de *slots* soporta el uso de objetos de predefinidos por el framework de forma transparente.

Además de los dos tipos de objetos documento descriptos, el programador de la aplicación puede hacer uso de objetos primitivos. Un conjunto de objetos Smalltalk pueden usarse como objetos primitivos, pero con una restricción mayor: El programador debe asegurar, que no ocurre ningún cambio a estos objetos fuera de una transacción.

Implementando Views

Una aplicación de groupware consiste de más de un espacio de objetos compartidos como describió anteriormente. Además de los objetos compartidos, hay objetos locales que representan visualización local y son responsables de la interpretación de eventos de entrada locales. Estos objetos también interactúan con objetos locales del sistema que representan la infraestructura local, como el sistema de ventanas o dispositivos de audio. Si alguna información representada en el espacio de objetos compartido se muestra en una visualización local, es responsabilidad de algunos objetos de la interface local de usuario transformar la información abstracta en una forma visual y aceptar la interacción gráfica referente a esta visualización.

Objetos locales que coexisten con objetos compartidos

Los objetos locales se ligan al ambiente local de un sitio, es decir, los dispositivos de la entrada y salida, de manera tal que no pueden ser parte del espacio compartido de objetos. Sin embargo, la necesidad de objetos locales genera interacciones entre objetos locales y objetos compartidos. Desde el punto de vista del flujo de información, estas interacciones pueden verse en dos direcciones:

1. El flujo desde los objetos locales, que interpretan la entrada del usuario, hacia los objetos compartidos. Para realizar este flujo de información, es necesario que los

objetos locales puedan hacer llamadas a los objetos compartidos. Esto no genera ningún problema.

2. Información que tiene que ser mostrada fluye desde los objetos compartidos hacia los objetos locales. Por consiguiente, parece ser necesario que los objetos compartidos pueden enviar mensajes a los objetos locales. Esto parece ser un problema, porque los objetos locales no tienen una identidad única globalmente y los objetos compartidos no pueden enviar mensajes a objetos que no conocen.

Para resolver este problema el framework COAST usa un sistema de restricciones de una dirección. En dicho sistema, el programador de la aplicación declara dependencias en lugar de programar métodos de actualización. Estas dependencias son llamadas *dependencias definidas por el programador de aplicación.* Una ventaja del sistema de restricciones es que el flujo de información de los objetos compartidos hacia los objetos locales puede realizarse sin enviar mensajes. Hay objetos locales que son dependientes de objetos compartidos, es la tarea del motor del sistema de restricciones es llevar a cabo el flujo de información. El sistema de restricciones organizará este flujo de información luego de que los objetos locales se han creado.

Restricciones inherentes del sistema

COAST define la presencia de un usuario como la relación entre un usuario y el sitio (workstation) en que el usuario está trabajando actualmente. Usando la información sobre presencia de usuarios, se definen las siguientes restricciones de sistema:

1. Para cada presencia de usuario, los canales de salida (ventanas, canal de audio, etc) tienen que estar abiertos para cada relación de interés entre el objeto compartido que representa al usuario (el modelo del usuario) y un objeto compartido que representa la aplicación (el modelo de la aplicación).

2. Cada canal abierto debe mostrar o modificar información actualizada, referenciada y definida por la aplicación en cualquier momento.

Estas restricciones deben mantenerse por un motor de constreñimiento que abre y cierra toma de los canales de salida y crea, elimina o actualiza objetos locales en respuesta al cambio de objetos compartidos.

Notificación del usuario

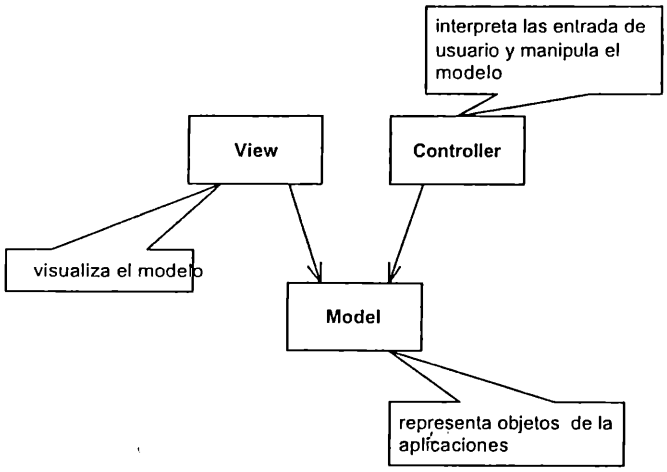
Gracias al sistema de constreñimiento descrito anteriormente, es posible notificar a otros usuarios sin la necesidad de usar un mecanismo de pasaje de mensajes interproceso o de eventos.

Una parte de información (ej. un recordatorio o mensaje) tiene que ser combinado con una aplicación para su visualización (ej. Sound player o ventana del mensaje), la misma es agregada a los objetos de interés del usuario. La notificación localizará al usuario lo más pronto posible (es decir en cuanto él se conecte al espacio del objeto compartido).

MVC

Los objetos GUI bajo COAST pueden ser implementados usando el constructor de interfaces de usuario modificado. Si el programador de la aplicación desea implementar sus propios objetos GUI, tiene que conocer a lo siguiente.

El framework COAST esta basado en el paradigma MVC (MVC = Model View Controller) esto significa que los objetos GUI son considerados como objetos view y controller. Los objetos view son responsables de mostrar un modelo (documento) en la pantalla. Los objetos controller se asocian a los objetos view y son responsables de interpretar la entrada interactiva de usuario. Como resultado de una interpretación, el controller envía a los mensajes al model object para causar un cambio en la estructura del documento.



En COAST, hay dos reglas principales, que el programador de la aplicación tiene que cumplir al programar las views y controllers:

- Los objetos view deben implementarse según un esquema de actualización especial, basado en la declaración de dependencias funcionales.

- Cuando los objetos controller interactúan con objetos del modelo (documentos) para provocar un cambio, los mensajes enviados tienen que ser encapsulados en una transacción.

El paradigma MVC Cooperativo de COAST, difiere del paradigma MVC convencional en muchos aspectos.

Cuando se implementan views bajo COAST:

- El programador de la aplicación no implementa ningún tipo de código responsable de su actualización (como el método *update:*).
- El programador de la aplicación no registra la view explícitamente para ser dependiente de otros objetos (ej. en objetos del modelo).
- El programador de la aplicación no invoca ningún tipo de mecanismo de actualización explícitamente (como por ejemplo, el métodos *changed:*).

En cambio, el programador de la aplicación divide sus variables de vista (slots) en dos categorías:

- Slots que contienen valores redundantes, que pueden computarse de otros valores (ej. contenidos en el modelo).

- Slots que contienen valores básicos, guardan informació n no redundante, que no depende de otros valores, por consiguiente no tiene que ser actualizada como reacción al cambio de otros slots.

Los slots de primer tipo serán declarados por el programador de la aplicación junto con un método que computa el valor del mismo. COAST asegurará que, en cualquier momento, el valor leído será un dato válido, de acuerdo al método que lo cómputo.

Este método puede ser un método formalizado, permitiendo al sistema realizar actualizaciones incrementales del valor, o un método Smalltalk arbitrario. En ambos casos, el sistema automáticamente rastrea todas las dependencias a otros slots, por eso es capaz de realizar actualizaciones de forma eficaz. En otras palabras, el sistema COAST le permite al programador de la aplicación asumir que la relación funcional, declarada por el método del cómputo, siempre será verdadera.

Un rasgo especial del sistema de views de COAST es también que el área del visualización de una view es considerada como un valor redundante, computado por el método " *displayOn:* ". Por eso, la visualización propia está actualizándose de la misma manera que los slots dependientes, sin ningún esfuerzo extra para el programador de la aplicación.

Esto produce una nueva filosofía para la programación de la visualización. No se provoca ningún efecto explícito en la visualización. Generalmente, los datos se muestran porque hay una visualización que consiste en ventanas, views y subviews y porque estos componentes dependen de los objetos compartidos que pueden haber cambiado. Cada view muestra datos en la pantalla, no porque el programador de la aplicación llama funciones de visualización, sino porque simplemente existe.

Por consiguiente, las ventanas en COAST no son abiertas explícitamente por el programador de la aplicación. La apertura de una ventana explícitamente sería un modo imperativo de causar efectos en la visualización. La idea fundamental es la presuposición que,

como todo los otros efectos en la visualización, una ventana está allí porque es implícita del modelo de la aplicación.

Implementando Controllers

Al implementar controllers bajo COAST:

- El programador de la aplicación no envía ningún mensaje (directamente o indirectamente) para modelar objetos sin haber empezado una transacción.
- El programador de la aplicación no espera por los eventos o pide una actualización de la visualización dentro de una transacción (esto significa que las transacciones son fine-grained y cortas).
- El programador de la aplicación no hace a ninguna asunción implícita si un objeto global ha cambiado o no, entre dos transacciones locales.

El sistema de transacción de COAST asegura la atomicidad y aislamiento de transacciones. La consistencia se asegura hasta donde expresó en las reglas de consistencia declaradas por el programador de la aplicación.

El manejador de transacciones de COAST puede procesar transacciones totalmente optimistas, sin retraso local. Las transacciones procesadas con optimismo que faltan son rolled back con retraso. El programador de la aplicación puede definir handlers para tratar los fracasos de una transacción. Sin embargo, el programador de la aplicación es responsable de prevenir cualquier tipo de dead locks que su código pueda causar.

El manejador de transacciones de COAST también proporciona transacciones no optimistas.

Principios del sistema COAST

COAST usa Frame System

COAST usa frames para modelar objetos del espacio de los datos compartido de la aplicación. Los frames se utilizaron originalmente dentro del dominio de inteligencia artificial para la representación de conocimiento. Las relaciones entre los frames no son como las referencias en los lenguajes de programación ordinarios. Ellos pueden llevar semántica como inversión y reglas de la conclusión.

Una meta principal de un frame system es agregar más semántica a las estructuras de los datos orientadas a objetos de una manera declaratoria. Hay semántica muy pequeña que el programador puede agregar normalmente a los atributos de un objeto. En lenguajes tipados, el tipo, o clase, de un atributo puede proporcionar alguna pista sobre las estructuras del objeto. En lenguajes no tipados, como Smalltalk, no hay ninguna semántica en absoluto, a pesar del nombre

del atributo, que puede ayudar a que las personas entiendan el propósito de un atributo. Un frame system le permite al programador de la aplicación declarar propiedades y restricciones, como inversión y cardinalidad, para los atributos del objeto. Estos atributos enriquecidos son llamados **slots** y los objetos usados en los slots se llaman **frames**. Un frame system consiste de todo lo necesario para declarar y usar frames.

Los *frame system monitors* aceden a los frames y slots para proporcionar las propiedades declaradas o restricciones de chequeo. Eso significa que el frame system tiene pleno control sobre el estado de todos los frames. Basado en este hecho, el frame system puede proporcionar medios genéricos adicionales como transacciones o el almacenamiento persistente que pueden usarse para cualquier frame definido por el usuario. El frame system de COAST usa herencia para obtener el control en todo los accesos a los slots.

COAST proporciona un espacio de datos compartido

El sistema COAST proporciona un espacio de datos compartido, donde pueden compartirse objetos específicos del modelo entre los usuarios localmente distribuidos. Dependiendo de la aplicación los espacios de datos compartidos pueden ir desde documentos (ej. una fracción de texto, un calendario, etc.) a espacios de información muy compleja (ej. un documento del hypermedia, un mundo virtual, etc.).

Técnicamente, hay muchas arquitecturas de distribución posibles, para realizar este espacio de datos compartido. En una arquitectura centralizada, los datos compartido se localizan en un servidor central, y los clientes tienen que enviar solicitudes al servidor cada vez ellos quieran accederlo. Como contraposición, en una arquitectura replicada, los clientes guardan copias locales de los datos compartidos, y un mecanismo de sincronización asegura que estas copias se mantienen actualizadas. Considerando que las arquitecturas centralizadas son más fáciles de implementar, las arquitecturas replicadas pueden proporcionar acceso inmediato a los objetos compartidos sin tener que esperar por demoras de la red.

Replicación y sincronización

Por razones de disponibilidad de datos y reaccion a las modificaciones, parte del espacio de datos es replicado por COAST. Los mecanismos de sincronización de COAST siempre aseguran un espacio de datos consistente, lo más pronto posible (con respecto a los atrasos de la red y la carga de procesamiento). La replicación y sincronización de los datos de la aplicación son completamente manejadas por COAST.

La unidad más pequeña de replicación es el *cluster*, una colección de model objects.

Estructuración del espacio de datos

Cómo estructurar el espacio de los datos compartidos depende del diseñador de la aplicación. COAST ofrece dos dimensiones de estructuración:

1. Los clusters como un grupo de model objects que se usan juntos.
2. La locación de almacenamiento del cluster, es decir dónde (en que el servidor) se guardará cada cluster.

Dependiendo de la complejidad del espacio de datos, la estrategia de distribución de los datos, y escalabilidad, los diseñadores de la aplicación pueden escoger usar uno o varios clusters distribuidos en uno o más servidores. Junto con la arquitectura flexible del sistema COAST compuesta de clientes, mediadores, y servidores que se apoyan muchos tipos diferentes de aplicaciones cooperativas.

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

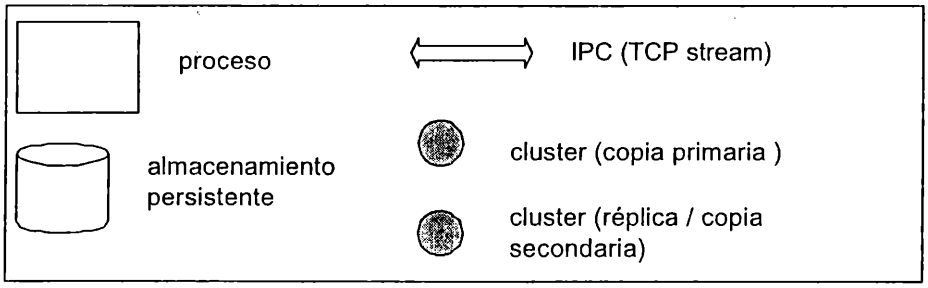
COAST emplea una arquitectura distribuida y replicada: Cada usuario interactua con una instancia individual de la aplicación, desarrollada en base a COAST, el *COAST Client*. Los clientes COAST normalmente proporcionan una interface al usuario final que permite manipular datos de la aplicación activamente.

COAST Mediator – sincronización y almacenamiento

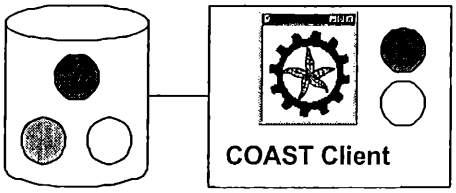
Los mediadores COAST negocian la comparición sincrónica de los datos de la aplicación. Ellos mantienen la copia primaria de los datos que son compartido entre sus clientes. Los datos de la aplicación necesarios para ejecutar una aplicación COAST son cargados y replicados transparentemente bajo demanda 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.

Ejemplos de la arquitectura

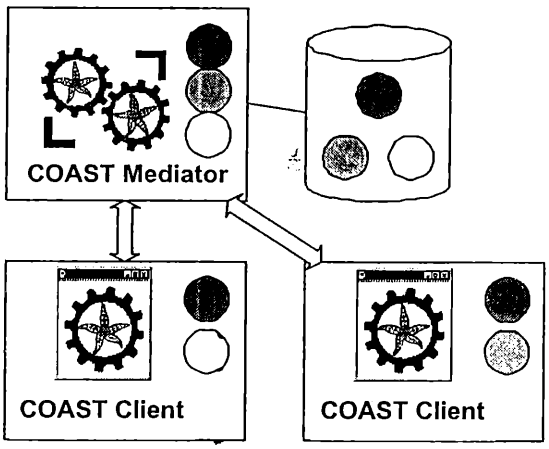
A continuación, se muestran tres arquitecturas de ejemplo. Las cajas grandes en los cuadros denotan sitios/procesos potencialmente diferentes en un ambiente de computadoras conectadas a una red. Las flechas denotan canales de comunicación entre procesos. Los círculos pequeños representan clusters donde, los rodeados con el mismo sombreado son réplicas diferentes del mismo cluster.



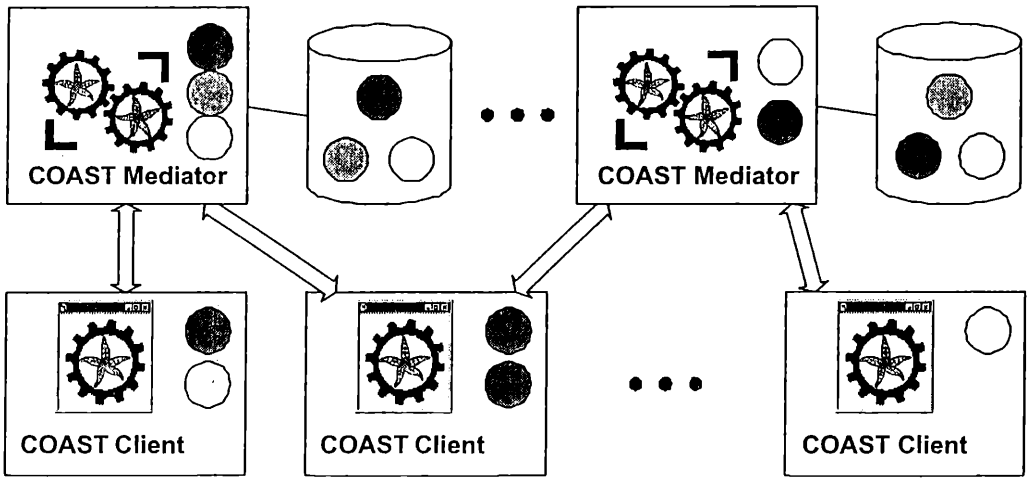
notación



Versión monousuario: adaptada para un solo usuario, ningún espacio de datos compartido.



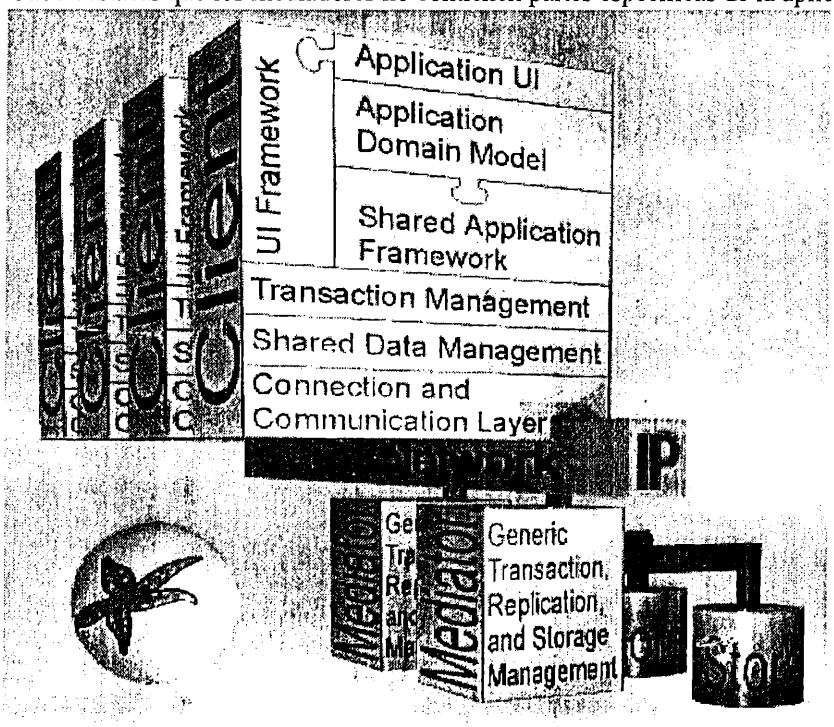
Versión multiusuario: adaptada para espacios de datos medianos y unos usuarios



Versión multiusuario: adaptada para espacios de datos grandes y muchos usuarios

El cuadro completo

El cuadro siguiente ilustra la arquitectura del sistema y la arquitectura de software de capas juntas. Aquí se pueden ver a los clientes gordos (lógica de la aplicación completa en el sitio cliente) con las partes de COAST genéricas, en amarillo, y las partes específicas de la aplicación, en verde. Cabe destacar que los mediadores no contienen partes específicas de la aplicación.



Capítulo 5

Objetivos de Chatblocks

La idea es aportar una arquitectura para la creación de herramientas chat, las ventajas principales que se desean brindar, en la creación de herramientas chats, son:

- **Facilidad**: Ayuda a que el costo de aprendizaje sea bajo, ya que si la arquitectura es lo suficientemente fácil de entender, los tiempos de aprendizaje serán más cortos lo cual lleva a un bajo costo económico.

- **Rapidez**: Baja el costo de implementación, si los tiempos de creación de las herramientas chat son cortos, el costo económico será bajo, permitiendo una rápida recuperación del tiempo que se necesita perder para el aprendizaje de la arquitectura. La idea es que al momento de crear un chat, el programador decida solo que componentes se desean usar. Al igual que lo hace, por ejemplo, en la creación de una interfaz gráfica al utilizar un editor de GUIs, el diseñador sólo selecciona los widgets que desea usar, no tiene que programarlos él.

- **Flexibilidad**: Permitir crear aplicaciones no convencionales.

- **Extensionalidad**: Por ser un framework, chatblocks permitirá futuras extensiones, aumentando el nivel de flexibilidad a la hora de crear una nueva herramienta chat.

- **Soporte para la implementación de protocolos explícitos**: Uno de los atractivos más importantes, de las herramientas chat que se podrán crear con Chatblocks, es el hecho de poder proporcionar protocolos explícitos, para ello se necesita soportar una serie de aspectos:

    1. *Floor Control.*
    2. *Clasificación de Usuarios.*
    3. *Numero de Participantes.*

Chatblocks es un framework

Un framework es un conjunto de clases que incorpora un diseño abstracto para solucionar una familia de problemas. Los frameworks son aplicaciones semicompletas, que pueden ser especializadas para producir aplicaciones específicas. Entre los beneficios que provee un framework orientado a objetos a los desarrolladores, encontramos: modularidad, reusabilidad, extensionalidad e inversión de control.

Lo que se desea con Chatblocks, es realizar una arquitectura que permita la creación, rápida y fácilmente, de una familia de herramientas chat, con características similares y algunos aspectos personalizados. La mayoría de la funcionalidad, de las aplicaciones finales, es la misma, por ejemplo el mecanismo de envío de mensajes es el mismo para todos las aplicaciones chats creadas con el framework, lo que puede llegar a cambiar es como se crean esos mensajes (si se les agrega semántica o no) y como se visualizarían. Por lo tanto la realización de un Framework es la solución que más se adapta a las necesid ades planteadas.

Chatblocks está basado en COAST

Chatblocks esta programado sobre un framework para groupware sincrónico (COAST), el mismo provee facilidad para la creación de aplicaciones cooperativas, así como también independencia del mecanismo de persistencia y comunicación subyacentes.

Las aplicaciones cooperativas sincrónicas (groupware sincrónico) permiten a un grupo de usuarios trabajar conjuntamente y simultáneamente en datos compartidos, como un documento. Un ejemplo podría ser, un sistema de reserva de vuelos, donde las acciones de cada usuario serán visibles al instante para cada uno de los demás usuarios de la aplicación. La aplicación que se está consideración aquí requiere un alto grado de sincronización, es decir, las acciones realizadas por un usuario deben ser percibidas inmediatamente por los demás usuarios cooperativos. No es suficiente observar los cambios realizados sobre la fuente de datos en común, también se necesita información sobre el cambio del contexto, considerando quién comenzó la acción, y si posible, por qué fue comenzada. Recíprocamente, los usuarios deben ser conscientes que sus propias acciones pueden tener efectos en otros usuarios del sistema. Esta percepción de qué están haciendo otros miembros del grupo es conocida como *group awareness* (conocimiento de grupo), y sirve para ayudar a que los usuarios coordinen sus acciones dentro del grupo. 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.

Aspectos a cubrir por ChatBlocks

El diseño de dicho framework deberá cubrir los siguientes aspectos:
Chatblocks cubre los aspectos mencionados en el capítulo1 de dos formas:

1. **Hotspots:** Aspectos personalizables en cada aplicación final. El programador de las aplicaciones chats creadas con Chatblocks podrá elegir, para cada uno de estos aspectos, entre las variantes proporcionadas por el framework.

2. **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.

Frozenspots

Unidad de Envío

- *Mensaje*

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 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.

Nro de Participantes

- *Dos o Mas*

Chatblocks esta pensado para la creación de chats grupales donde existen mas de dos participantes, el numero de participantes podrá ser asignado por parámetro al momento de la creación del chat.

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.

Receptor

- *Todos*

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 un 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.

Formas de Recepción

- *Todos los mensajes emitidos*

Ya que el framework permitirá la realización de chat tanto sincrónicos como asincrónicos, debe proveer un mecanismo de recepción acorde con estos requisitos.

Los usuarios recibirán todos los mensajes que sean enviados, tanto los que se envían mientras el usuario esta conectado, como los enviados en los períodos de desconexión del usuario.

Hotspots

Desarrollo de la conversación

- *Sincrónica*
- *Asincrónica*

El framework intenta abstraer al programador de lidiar con protocolos de red, comunicación, mecanismos de sincronización y demás problemas existentes en las aplicaciones chat. Para ello se implementarán dos modalidades de desarrollo de la

conversación, sincrónica y asicrónica, permitiendo al programador seleccionar que modalidad desea obtener con respecto a este aspecto.

Clasificación de Usuarios

- *Sin Roles*
- *Con Roles*

El framework permitirá soportar usuario con diferentes roles o que ninguno cumpla un rol especial dentro de la aplicación (SinRoles). 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 asumen los usuarios.

Floor Control

- *Ninguno*
- *Protocolo*

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 lá 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.

La mayoría de las aplicaciones cooperativas (groupware) poseen Floor Control, pero los chats comerciales no. Esto se debe a una falta de investigación en el área, lo cual lleva a una falta de innovación en este tipo de herramientas.

Continuando con el ejemplo de un juego de ajedrez, la aplicación fuerza a los usuarios a seguir un protocolo explícito, ya que cuando un jugador esta realizando su movida, el otro debe esperar su turno.

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 mensajes 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.

Criterio de Ordenación

- *Atributos*
- *Semántica*

Los mensajes podrán ser ordenados tanto por algún atributo del mismo o por su semántica. En el caso de atributos, estos pueden ser su autor o su fecha de creación. En cuanto a la semántica, el criterio podrá ser el thread de conversación al que pertenecen o por las referencias a los objetos.

Visualización de los mensajes

- *Arbol*
- *Lista*

El framework proveerá dos modalidades a seleccionar, 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.

Awareness

- *De Participación*
- *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.

Representación del autor del mensaje

- *Nombre*
- *Imágenes*
- *Tipo y color de Font*

Las forma de representar al usuario que proveerá el framework podrán ser textuales o gráficas. En el primer caso, con el nombre del usuario emisor del mensaje o con el tipo y color de font, En el caso de representación gráfica, con una imagen (icono).

En este caso se proveen estas tres modalidades, ya que no existe una que sea "la mejor opción".

Post-Edición

- *Borrado*
- *No Permite*

Chatblocks proveerá la edición de los chats, la modalidad elegida para este caso es el borrado 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.

El programador también podrá optar por no permitir el borrado de los mensajes.

Formato de Envío

- *Texto Plano*
- *Texto con Formato*

El framework proveerá dos tipos de formatos de envío, por un lado texto plano, esto es útil cuando el tipo y color de font conforma la representación del usuario. La

segunda opción que proveerá el framework será Texto con formato, este permite destacar ciertas palabras en el texto.

Filtros

- *Emisor*
- *Contenido*
- *Thread*

Con respecto a este aspecto, el framework proveerá tres modalidades.

Los mensajes podrán ser filtrados por diferentes atributos, ya sea su autor, contenido. En cuanto a los mensajes que poseen semántica, de estos se podrán elegir todos los que pertenezcan a un mismo thread de conversación.

Extensiones

Las aplicaciones creadas con Chatblocks están diseñadas para ser subaplicaciones 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.

Capítulo 6
El Framework: Chatblocks

Chatblocks es un Framework Orientado a Objetos, de caja blanca y dominio especifico. El dominio del framework es la conversación mediante el uso de computadoras, éste provee soporte para la creación de herramientas chat de forma rápida y precisa, ya que permite al usuario del framework, el programador de la aplicación, personalizar varios aspectos de la aplicación a construir. Por ser un framework de caja blanca, la extensión se realiza por medio de subclasificación de dases existentes, permitiendo al usuario del framework extender o agregar componentes.

La idea de Chatblocks es aportar herramientas para la creación de aplicaciones chat, capaces de representar diferentes situaciones de diálogo. Esto se obtiene a partir de una estructura y distintos componentes.

La estructura, forma el esqueleto de la aplicación, provee la funcionalidad común a todas las aplicaciones del dominio y en determinados lugares existen espacios en blanco. Cada espacio requiere de una funcionalidad especifica la que es ejecutada por un tipo de componente.

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.

Chatblocks se compone, por ser una aplicación COAST, 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:

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

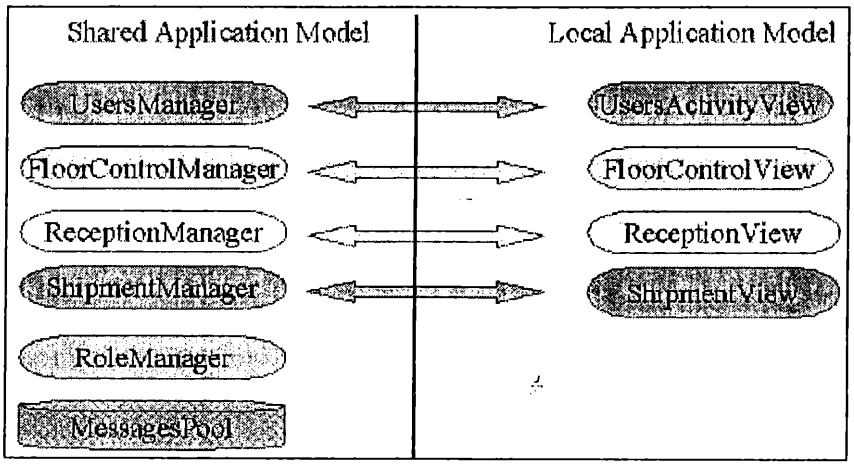
Como lo indica la tabla, en la parte compartida se encuentra el MessagesPool y los managers, y en la parte local las views.

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 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.

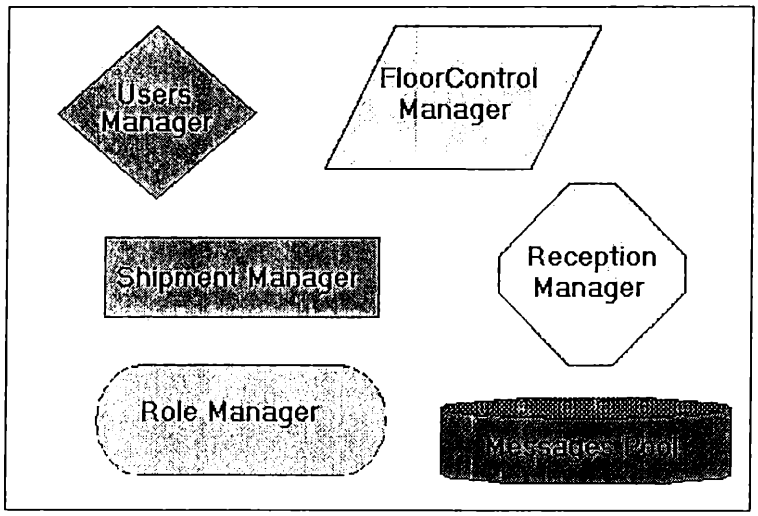
La interfaz del usuario surge de la composición de views, mientras que el corazón de la aplicación, la parte compartida, surge de la composición de managers y el MessagesPool.



Componentes del Framework

Shared Application Model

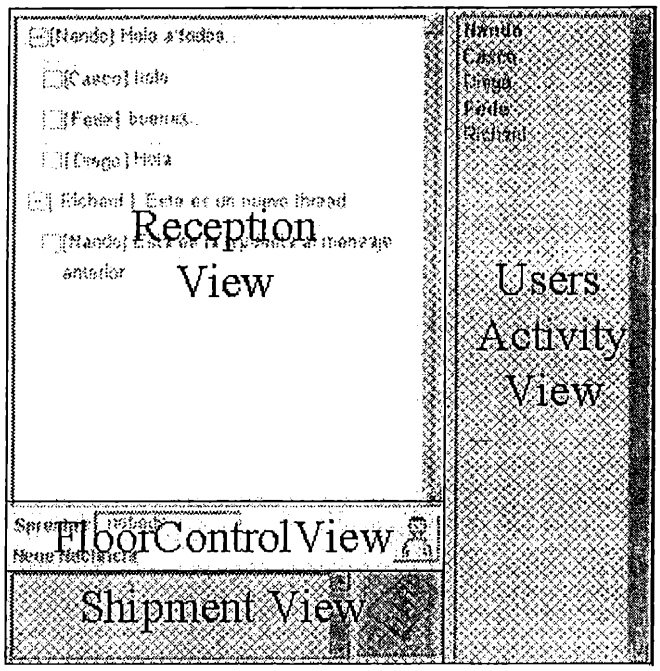
Componente	Tipo de Clase
UsersManager	Concreta
ReceptionManager	Concreta
ShipmentManager	Abstracta
FloorControlManager	Abstracta
RoleManger	Abstracta



Estructura de la parte compartida de la aplicación

Local Application Model

Componente	Tipo de Clase
UsersView	Concreta
FloorControlView	Abstracta
ShipmentView	Abstracta
ReceptionView	Concreta



Estructura de la parte local (interfaz de usuario)

Las clases abstractas, indican que a partir de ellas se forma una familia de componentes, estas poseen el comportamiento general del componente pero en la aplicación real (chat), se deberá optar por una de sus subclases concretas.

Las clases concretas, son componentes comunes a todas las aplicaciones chat que se creen con el framework, estos componentes no se pueden cambiar, pero admiten cambios en su configuración, proveyendo flexibilidad, pero en un grado menor que los anteriormente mencionados.

Cabe destacar que el hecho de que estos componentes sean clases concretas no impide que en un futuro estas se conviertan en abstractas y así poder obtener aun más flexibilidad. La razón de estas es que con este primer paso del framework necesitaba obtener la mayor flexibilidad posible en los Roles de Usuario y Floor Control. Estos son dos aspectos muy importantes en la conversación, se necesita proveer la suficiente flexibilidad para poder recrear las situaciones de la vida real y los protocolos existentes. Estos aspectos, forman un área de estudio que carecen de experimentación y por lo tanto no han evolucionado como lo han hecho otros aspectos dentro de este tipo de aplicaciones.

En la parte local de aplicación se encuentra la interfaz de usuario, esta es independiente por cada usuario, es decir, existe una por usuario. Esto permite la personalización de la aplicación por parte de los usuarios, cabe destacar que, si bien todos los usuarios comparten los objetos del

corazón de la aplicación, SharedApplication, esto no implica que a todos les interese la misma información, o la deseen ver la misma forma, por ese motivo el framework provee mecanismos de personalización por usuario.

Domain Model

MessagesPool

Es el repositorio de mensajes. Esta clase es una clase concreta, su función es mantener todos los mensajes que se intercambian a lo largo de la conversación.

Cuando un mensaje es enviado el ShipmentManager lo agrega al MessagesPool, el ReceiverManager retira todos los mensajes de este repositorio cuando debe entregar los mensajes al usuario.

Esta clase consta de una colección, donde se almacenan los mensajes, llamada *messages,* y las operaciones necesarias para la manipulación de los mismos, es decir, agregar eliminar y buscar mensajes entre otras.

Message

Representa un mensaje. Esta es una clase concreta, pero también existen dos subclases, cada una de ellas aporta semántica a los mensajes. Esta clase posee tres atributos principales:

- *Author:* Referencia a un ChatUser, que es el emisor del mensaje.
- *Text:* Es un String, el mensaje en sí.
- *CreationTimestamp:* Fecha y hora de creación.

Si bien esta es una clase concreta, existen dos subclases, también concretas. Estas agregan atributos y sus respectivas operaciones.

Subclases

ThreadReferenceMessage

Incorpora el atributo *parent*. Este es una referencia a un *ThreadReferenceMessage*, indicando que un mensaje es la respuesta a su *parent*, la cadena de referencias entre mensajes forman los thread o hilos de conversación.

ObjectReferenceMessage

Incorpora el atributo *reference*. Este es una referencia a un objeto cualquiera, en este caso no existen restricciones sobre clases.

ChatUser

Representa al usuario del chat.

Posse una imagen (icon), la cual se puede utilizar como avatar y una instancia de UserRole, que representa e rol del usuario en el chat.

La forma de asignación de los roles puede ser aleatoria o predefinida, teniendo para esto un diccionario con roles y usuarios a los que se les deben asignar esos roles.

Metodos

role: aUserRole
Recibe como parametro un UserRole y lo asigna como rol actual del usuario.

role
Retorna el rol actual del usuario.

UserRole

Es una clase abstracta.

La clase UserRole representa un rol de un participante del chat, dependiendo del tipo de conversación que queremos representar, debemos modelar diferentes familias de roles de usuarios. El framework provee tres tipos, que se detallan a continuación, en caso de querer agregar una familia de roles o un rol simple, solo se debe con agregar una subclase concreta de UserRole y una subclase concreta de RoleManager que asigne estos nuevos roles de usuario.

Roles implementados

NoRoleUserRole: Representa un rol nulo, se debe usar este rol cuando no se necesitan dietinguir entre diferentes usuarios. Por ejemplo, este rol se debería usar para realizar un chat convencional (ChatRooms de la web), donde no existe diferenciación entre roles de usuario.

DiscussionUserRole: Es una clase abstracta, forma una familia de roles, que modela los roles que existen en una discusión, donde existe un mediador y varios participantes. De esta se desprenden dos subclases concretas:

- **DiscussionMediatorUserRole:** Es el mediador de la conversación.
- **DiscussionParticipantUserRole:** Es un participante del chat.

ProContraUserRole: Es una clase abstracta, forma una familia de roles, que modela los roles que existen en una discusión, donde existen dos partes, una pro y otra contra, los cuales discuten

una idea, y los demas participantes forman la audiencia del debate. De esta se desprenden tres subclases concretas:

- **ProUserRole:** Es una de las partes del debate.
- **ContraUserRole:** Es la otra parte del debate.
- **HearerUserRole:** Es un oyente del debate.

Shared Application Model

ChatSharedApplication

Conforma la parte compartida del chat. Esta es una clase abstracta, ya que solo sirve de esqueleto para las futuras aplicaciones. Esta clase posee seis atributos, que son slots Coast, cada uno de estos referencia a un componente del chat. Para personalizar estas referencias solo basta con crear una subclase de ChatSharedApplication y reescribir los métodos de iniciación de los slots a personalizar.

El framework provee, en algunos casos varios componentes para cumplir una determinada función:

Función	Componente concreto
MessagesPool	MessagesPool
UsersManager	UsersManager
ReceptionManager	ReceptionManager
RoleManager	NoRoleRoleManager ProContraRoleManager DiscussionRoleManager
ShipmentManager	NoReferenceShipmentManager ThreadReferenceShipmentManager ObjectReferenceShipmentManager
FloorControlManager	NoProtocolFloorControlManager ProContraFloorControlManager AutoMediatorDiscussionFloorControlManager HumanMediatorDiscussionFloorControlManager

UsersManager

Forma parte del Shared Application Model, es una clase concreta.

Mantiene información sobre los participantes del chat, se encarga de asignar los avatars de los usuarios a los mensajes.

Mantiene una colección con todos los participantes de la conversación (ChatUsers) y otra con los participantes que se encuentran conectados (OnLine).

Métodos

login: aChatUser
Recibe como parámetro un ChatUser y lo agrega a la colección de conectados (on-Line).

logout:aChatUser
Recibe como parámetro un ChatUser y lo elimina de la colección de conectados (on-Line).

onLine: aChatUser
Recibe como parámetro un ChatUser y retorna un boolean, true si el user esta on-Line.

maxParticipants: aNumber
Asigna el numero máximo de participantes del chat. Inicialmente es infinito (0).

minParticipants: aNumber
Asigna el numero mínimo de participantes del chat. Inicialmente es 2.

participants
Retorna una colección de usuario, los participantes de la conversación.

onLineParticipants
Retorna una colección de usuario, solo los que están on-LIne.

RoleManager

Forma parte del Shared Application Model, es una clase abstracta.

El RoleManager es el encargado de asignar los roles a los usuarios, estos son subclases de la clase UserRole, que es una clase abstracta.

Las subclases de UserRole forman familias de roles de usuario.

Para cada familia de roles existirá un RoleManager especifico, subclase de RoleManager, por ejemplo, el DiscussionRoleManager asigna roles del tipo DiscussionUserRole.

La forma de asignación de los roles puede ser aleatoria o predefinida, teniendo para esto un diccionario con roles y usuarios a los que se les deben asignar esos roles.

Métodos

setRole: aChatUser
Recibe como parámetro un ChatUser y a este le asigna un UserRole, mediante el método role: aUserRole.

El framework provee tres componentes RoleManager concretos:

- **DiscussionRoleManager**: Asigna a cada usuario un rol del tipo **DiscussionUserRole.**
- **NoRoleRoleManager**: Asigna roles **NoRole.**
- **ProContraRoleManager**: Asigna a cada usuario un rol del tipo **ProContraUserRole.**

FloorControlManager

El FloorControlManager es el encargado de mantener el control de piso de la aplicación, determinar que usuarios están habilitados para realizar operaciones. Cada protocolo que se desee modelar implica un floor control diferente.

En el caso de necesitarse un protocolo diferente a los que provee Chatblocks, este puede agregarse al framework. Para esto se deberá crear una subclase de FloorControlManager. Por lo general, cada protocolo necesita una familia de roles de usuario, por eso para cada implementación de protocolo, se deberá implementar una familia de roles y su respectivo RoleManager.

Al FloorControlManager le llegan los evento de entrada y salida de los usuarios al chat (estos los envía el UsersManger) y envíos de los usuario (capturado por el ShipmentManager). Para agregar la captura de otros eventos, se debe modifica el manager correspondiente, que sería el encargado avisarle al FloorControlManager del evento ocurrido.

Métodos

isAllowedToWrite: aChatUser
Recibe como parámetro un ChatUser y retorna un boolean, indicando si el usuario es autorizado a escribir.

login: aChatUser (Evento de ingreso de un usuario al chat)

Recibe como parámetro un ChatUser.

logout: aChatUser (Evento de egreso de un usuario al chat)
Recibe como parámetro un ChatUser.

messageSent: aChatUser (Evento de envío de un mensaje)
Recibe como parámetro un ChatUser.

FloorControlManager es una clase abstracta, de esta se desprenden cinco subclases, de las cuales cuatro son componentes concretos. Chatblocks implementa cuatro tipos de protocolos.

Subclases (Protocolos implementados)

NoProtocol (NoProtocolFloorControlManager)

Representa la ausencia de protocolo explícito en el chat, todos los usuarios pueden escribir en cualquier momento, no existen restricciones de ningún tipo, por lo tanto no se necesita llevar registro de las acciones que los usuarios.

Este protocolo, no requiere que los tengan asignados roles, por no tener restricciones sobre los mismos. Por lo tanto se utiliza la familia de roles **NoRoleUserRole**

Métodos

isAllowedToWrite: aChatUser
Retorna siempre true.

ProContra (ProContraFloorControlManager)

Representa un debate entre dos personas, sobre un tema determinado, donde uno tiene una posición a favor de tema (Pro) y otro en contra (Contra), la discusión se lleva a cabo de la siguiente manera.

Al comenzar la aplicación, la primera palabra la pueden tener tanto el Pro como el Contra, a partir del primer mensaje se produce una asignación de turnos, donde el autorizado a hablar es el que no hablo en el turno anterior.

Los demás participantes pueden realizar acotaciones, luego de que se halla enviado el primer mensaje, en cualquier turno.

Esta clase posee un atributo Turno, que indica a quien le toca hablar en el turno actual.

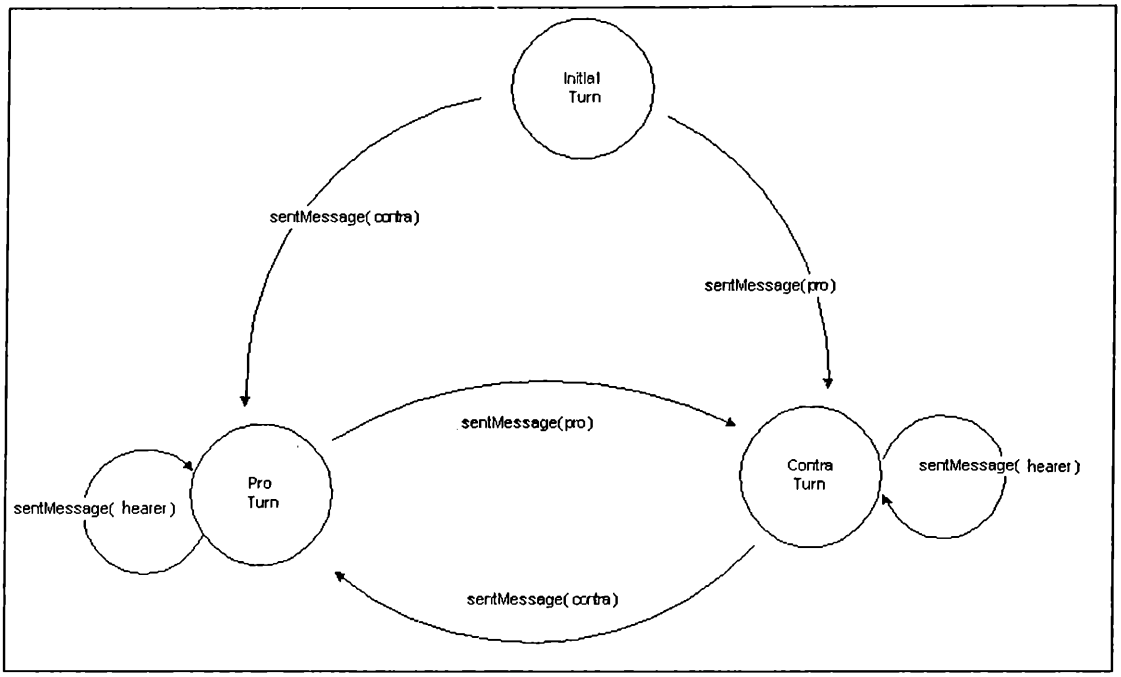


Diagrama de transición de estados del protocolo ProContra.

Métodos

isAllowedToWrite: aChatUser
si el rol de aChatUser es Hearer y el turno no es el inicial, retorna true.
sino Retorna el resultado de la expresión: turno = rol de aChatUser.

messageSent: aChatUser
Si aChatUser es pro o contra, cambia el turno.

Discussion (DiscussionFloorControlManager)

Emula el protocolo que existe en una conferencia de prensa o en una asamblea, donde existe un mediador y varios participantes de la conversación, la misma se desarrolla de la siguiente forma:

Un participante pide la palabra, levantando la mano, y el mediador selecciona uno de los que levantan la mano y le sede la palabra, este realiza su aporte y luego el mediador vuelve a seleccionar entre los participantes con la mano levantada uno.
se tienen tres estados
allowedToWrite : Participantes que están autorizados a hablar.

WaitingToSpeech : Participante con la mano levantada.
Listen : Participantes escuchando.

Cada participante esta solamente en uno de estos estados en un instante de tiempo.
Para llevar control de los estado de los participantes, se tienen tres colecciones: allowedToWrite, waitingToSpeech y listen.

Este protocolo requiere que los usuarios tengan asignados roles de la familia **DiscussionUserRole,** para lograr este resultado, el componente que debe ser usado como RoleManager de la aplicación es **DiscussionRoleManager.**

DiscussionFloorControlManager es una clase abstracta. De esta se desprenden dos subclases concretas, ambos componentes que proveen el protocolo de discusion o conferencia de prensa:

- **HumanMediatorDiscussionFloorControlManager**
- **AutoMediatorDiscussionFloorControlManager**

La diferencia entre estos dos componentes radica en que el primero requiere que exista un mediator, su rol será **DiscussionMediatorUserRole,** entre los usuarios del chat y los demas sean participantes, cuyo rol debe ser **DiscussionParticipantUserRole.** Por otra parte, el segundo componente requieren que el rol de todos los usuarios del chat sea **DiscussionParticipantUserRole.**

En cuanto a la ejecucion del protocolo, cada usuario esta habiliatado a realizar cierto tipo de operaciones, las mismas dependen del rol que posea. Existen operaciones que solo puede realizar el mediador y otras permitidas solo a los participantes. La diferencia en este sentido es, cuando hay varios participantes esperando el turno para hablar, el primer componente (**HumanMediatorDiscussionFloorControlManager**) espera que el mediador del chat le de el turno a alguno, en cambio el segundo componente (**AutoMediator...**) le da la palabra al primer usuario que la solicito, la estructura utilizada para este fin es una colección FIFO.

Métodos (Implementados en DiscussionFloorControlManager)

Operaciones de los participantes

Request: aParticipant

Un participante pide el turno para hablar. Emularia la accion de levantar la mano en un aula, o conferencia.

CancelRequest: aParticipant

Un participante que está pedidiendo turno para hablar, cancela el pedido. En este caso el usuario bajaría la mano sin poder hablar.

CancelWrite: aParticipant
Un participante actualmente posee la palabra,

Métodos (Implementados en HumanMediatorDiscussionFloorControlManager)
Operaciones del Mediador

SetTurn: aParticipant
AParticipant se recibe la palabra para hablar. Los requisitos para que se efectúe la operación son: Ningún participante tiene la palabra, aParticipant debe estar en la lista de espera de la palabra.

Al realizar determinadas operaciones sobre la aplicación, los usuarios del mismo, cambian de estado. Cada estado permite realizar ciertas operaciones.

Los estados del participante son los anteriormente mencionados: **AllowedToWrite, WaitingToSpeech** y **Listen.** Los estados del mediador son solo dos: **GiveTurn,** indica que debe dar la palabra a un participante, y **WaitingToGiveTurn,** que indica que debe esperar hasta que se cumpla la condición de dar la palabra (requisitos de la operación setTurn).

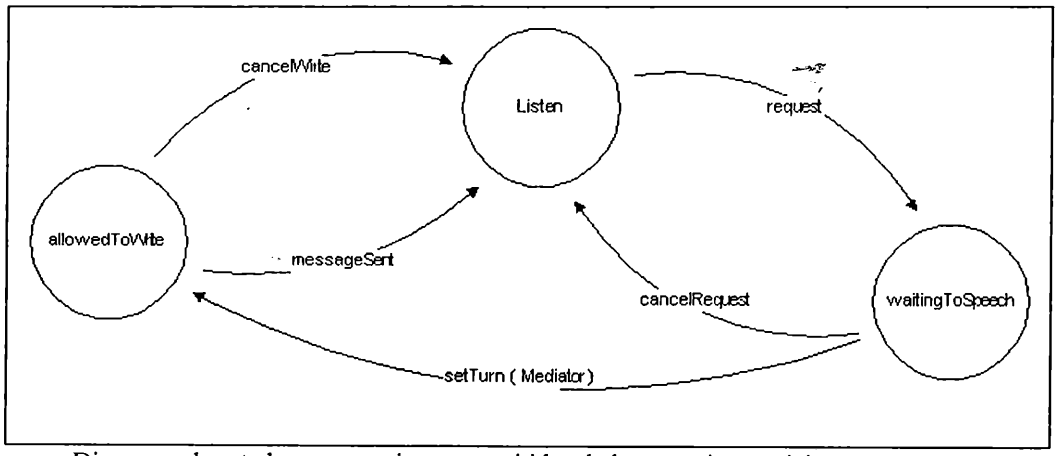


Diagrama de estados y operaciones permitidas de los usuarios participantes.

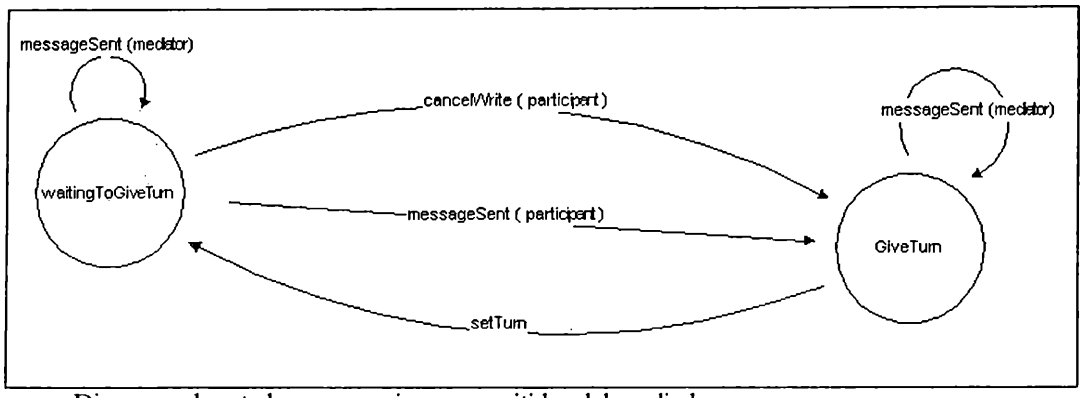


Diagrama de estados y operaciones permitidas del mediador.

Shipment Manager

Forma parte del Shared Application Model, es una clase abstracta.

De esta clase se desprenden tres subclases concretas, que conforman las tres variaciones que provee el framework para utilizar como ShipmentManager.

Este componente es el encargado de realizar las operaciones de envío de los aportes de los usuario, por medio del método **send.** La operación consiste en tomar los mensajes, que son solo strings, y crear un objeto del tipo **Message**, y luego agregarlos al MessagesPool, de donde luego se tomaran los mensajes para su lecutra.

En cuanto al mensaje, éste puede ser una instancia de la clase **Message** o alguna subclase de la misma, todos los componentes concretos responden al send, creando mensajes de la clase Message, luego cada uno agrega operaciones para agregar semántica a los mensajes, creando instancias de las subclases de **Message**.

Métodos

send(aString)

Crea una instancia de Message, asigna aString como el *text*, el smisor pasa a ser el *author*. Luego lo incluye al MessagesPool.

Subclases (Protocolos implementados)

NoReferenceShipmentManager

Envía mensajes simples, sin referencia, esto implica que los mensajes no poseen semántica. Sólo responde al send de la superclase.

ThreadReferenceShipmentManager

Envía mensajes que referencian a un "padre" o "antecesor" en un thread o hilo de conversación, los threads se van armando con las referencias de mensajes a otros. Estos forman una estructura jerárquica, ya que varios mensajes pueden referirse a un mismo mensaje.

Los mensajes sin referencia a un padre crean un thread, esto quiere decir que introducen un subtema a la conversación, en cambio los que referencian a un padre están continuando un thread.

Métodos

sendRefThread(aString, aMessage)

Crea una instancia de ThreadReferenceMessage, al atributo *parent* le asigna aMessage, aMessage puede ser de un ThreadReferenceMessage. Esta operación continua un Thread.

send(aString)

Realiza un forewardeo al mensaje sendRefThread(aString, aMessage), donde el string es el mismo, pero aMessage es Nil. Esta operación crea un nuevo Thread.

ObjectReferenceShipmentManager

Envía mensajes que referencian a un objeto; en este caso los objetos pueden ser de cualquier tipo, ya que la idea es que sea una referencia genérica. En caso de necesitar referencias de un tipo de objeto dado se puede subclasificar esta clase y aplicarle las restricciones del caso.

Es necesario aclarar este componente también responde al método send, por lo tanto es capaz de enviar mensajes sin referenciar ningún objeto.

Métodos

SendRefObject(aString, aObject)

Crea una instancia de ObjectReferenceMessage, al atributo *reference* le asigna aObject, aObject puede ser cualquier tipo de objeto.

ReceptionManager

Es una clase concreta. Este componente es el único que provee el framework para cumplir esta función. Es responsable de la recepción de mensajes y del filtrado de los mismos.

La recepción de los mensajes consiste en tomar los mensajes del MessagesPool, estos pueden ser filtrados o no.

Para el filtrado de los mensajes existe un subcomponente, llamado **MessageFilter** cuyo objetivo es reconocer, en una colección de mensajes dada, cuáles son los que cumplen con determinado criterio. Este subcomponente es referenciado por el atributo *filter*, este es un slot personalizado o PersonalizedValueHolder, esto quiere decir que cada usuario posee un objeto diferente como atributo, es necesario para dar independencia al filtrado de información ya que a cada usuario les interesa realizar filtros por diferentes criterios.

Métodos

GiveMessages
Retorna todos los mensajes que existen en MessagesPool.

GiveMessagesFiltered: aFilterParameter
Para realizar esta operación el atributo *filter* no debe ser nulo.
Toma la colección de todos los mensajes que existen en MessagesPool, luego invoca el método filterCollectionBy con la colección y aFilterParameter y retorna la colección resultante de este ultima operación.

SetFilterbyAuthor
Asigna una instancia de **MessageFilterByAuthor** al atributo *filter*.

SetFilterbyThread
Asigna una instancia de **MessageFilterByThread** al atributo *filter*.

SetFilterbyDate
Asigna una instancia de **MessageFilterByDate** al atributo *filter*.

El Framework provee tres subcomponentes **MessageFilter.** Los tres responden al método **filterCollectionBy(collectionToFilter, parameter)** para cada subclase el criterio de filtrado y el parámetro será diferente como se detalla a continuación.

- **MessageFilterByAuthor:**

Parameter: debe ser un ChatUser.
Retorna solo los mensajes cuyo autor sea el recibido como parámetro.

- **MessageFilterByDate**
  Parameter: debe ser un Date.
  Retorna solo los mensajes cuya fecha de creación sea igual a la recibida como parámetro.

- **MessageFilterByThread**
  Parameter: debe ser un Message.
  Retorna solo los mensajes que pertenezcan al mismo thread al que pertenece el recibido como parámetro.

En caso de necesitar un nuevo criterio para el filtrado de mensajes se podrá crear una subclase de MessageFilter e implementar el método filterCollectionBy(collectionToFilter, parameter).

Local Application Model

ChatLocalApplication

Es la contraparte local de ChatSharedApplication, esta forma la interfaz de usuario. Esta es una clase abstracta, y posee cuatro atributos, que son slots Coast, cada uno de estos referencia a un componente de la interfaz. Los componentes locales son elementos gráficos que al combinarlos forman la interfaz de usuario, cada tipo componente tiene su lugar asignado en la interfaz. El framework provee, en algunos casos, varios componentes para una determinada ubicación:

Ubicación	Componente concreto
UsersActivityView	UsersActivityView
ReceptionView	ReceptionView
ShipmentView	NoReferenceShipmentView ThreadReferenceShipmentView ObjectReferenceShipmentView
FloorControlView	NoProtocolFloorControlView ProContraFloorControlView ParticipantDiscussionFloorControlView MediatorDiscussionFloorControlView

Estos componentes reflejan los cambios en el *DomainModel*, cuando se enteran de un cambio le piden al componente correspondiente del *SharedApplicationModel* que les retorne la información para luego refrescar la interfaz de usuario.Para conocer el momento en que se producen los cambios en el *DomainModel* estos poseen **slots computados**. Este tipo de slot es consta de un bloque de ejecución, el valor de slot surge de la ejecución del bloque. La particularidad esta dada por el momento en que se ejecuta dicho bloque, para ello existen *demons* que están mirando constantemente cada uno de los objetos referenciados por el bloque,

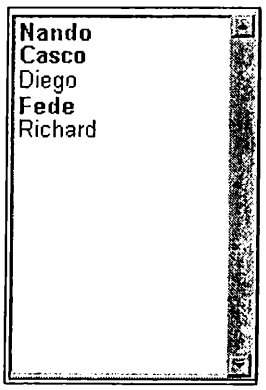
cuando uno de estos *demons* detecta un cambio en un objeto, quiere decir que el valor del slot cambio, por lo tanto se ejecuta el bloque, obteniendo como resultado la recomputacion de slot.

UsersActivityView

Esta es una clase concreta, exhibe la información de awareness que manipula el UsersManager: Esta componente consta de una lista donde muestra todos los usuarios que participan de la conversación, donde los usuarios conectados (On-Line) se muestran con el estilo de fuente bold y los desconectados con estilo normal.

Para realizar esta función posee dos slots computados, uno es la colección de participantes y el otro consiste en la colección de usuarios On-Line.

Este componente es el único que provee el framework para exhibir esta información, como idea a realizar en una futura extensión del framework es agregar un componente que muestre los iconos de cada usuario, en lugar de sus nombres.



UsersActivityView

ReceptionView

Es responsable de mostrar los mensajes que recibe un usuario.

A su vez esta compuesto por dos subcomponentes *viewer* y *ordered*. Estos son instancias de las clases **MessageViewer** y **MessageOrderer** respectivamente. El primero aporta la representación gráfica, mientras que el segundo agrega lógica para la ordenación de los mensajes.

Los mensajes son solicitados al ReceptionManager, este los entrega en una colección, la misma puede estar filtrada o no, esto depende del método que se utilice para solicitar los mensajes. Una vez obtenidos los mensajes, la colección que los contiene pasa por el *orderer,* que

ordena la colección dependiendo por un criterio determinado, dependiendo de la clase a la que pertenezca la instancia del mismo.

Luego del ordenamiento de la colección de mensajes esta pasa al *viewer,* donde se muestran al usuario, la forma de visualización será en forma de árbol o lista, dependiendo de la clase instanciada para realizar este trabajo.

Como *orderer* es posible utilizar cualquiera de las subclases de **MessageOrderer,** estas son:

- **MessageOrdererByThread:** Ordena los mensajes de acuerdo al hilo de conversación al que pertenezcan.
- **MessageOrdererByAuthor:** El criterio de ordenación en este caso es el autor o emisor del mensaje.
- **MessageOrdererByDate:** Los mensajes son ordenados por su fecha de creación de menor a mayor valor.

Como *viewer* es posible utilizar cualquiera de las subclases de **MessageViewer,** estas son:

- **MessageViewerList:** Muestra los mensajes en forma de lista.
- **MessageViewerTree:** Muestra los mensajes en forma de arbol. Los mensajes que posean algún atributo en común, dependiendo del criterio de ordenación utilizado, formarán una rama del árbol.

Los *viewers* se pueden combinar con los *orderers*, obteniendo como resultado nuevas formas de visualización de mensajes.

Métodos

ViewAsTree
Asigna una instancia de MessageViewerTree como *viewer.*

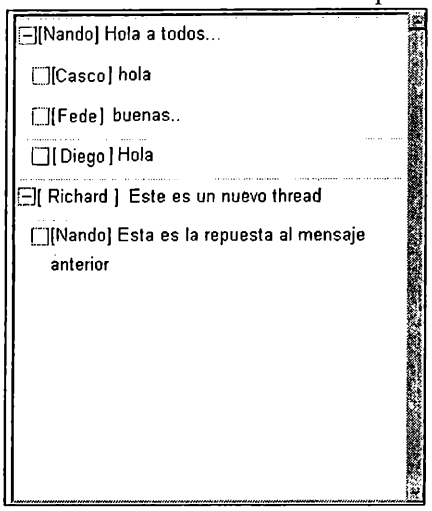
ViewAsList
Asigna una instancia de MessageViewerList como *viewer.*

OrderByDate
Asigna una instancia de MessageOrdererByDate como *orderer.*

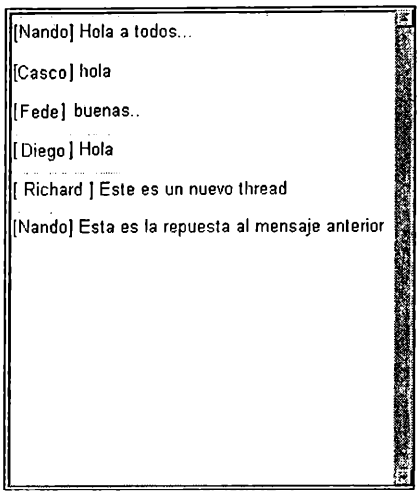
OrderByAuthor
Asigna una instancia de MessageOrdererByAuthor como *orderer.*

OrderByThread
Asigna una instancia de MessageOrdererByThread como *orderer.*

ReceptionView:Utilizando dos viewers diferentes.



MessageViewerTree



MessageViewerList

ShipmentView

Este componente es el encargado de despachar los mensajes al **ShipmenteManager** para que este último complete la operación.

Esta una clase abstracta, y de ella derivan tres subclases concretas:

La interfaz de usuario consta de un campo de texto, donde se escribe el mensaje a enviar, y un botón, que es el que realiza la operación de envío.

El envío esta orientado a mensajes, ya que el usuario primero lo escribe en el campo de texto y luego lo envía presionando el boton *send*, o la tecla *enter*.

El formato de los mensajes es texto plano, ya que esta versión del framework no provee asignación de estilos y tipos de fuentes a los mensajes.

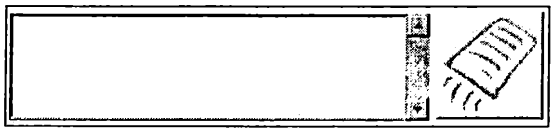
Este componente posee lógica para habilitar y deshabilitar el botón *send*, con la finalidad de indicar que el usuario puede enviar mensajes o no. Estas funciones están implementadas para brindarle apoyo al *FloorControlView*.

- **ThreadReferenceShipmentView**: Al enviar un mensaje interactua con el *ReceptionView*, y si existe algún mensaje seleccionado, el nuevo mensaje se transforma un la respuesta al mensaje seleccionado. Si no existe selección el mensaje se convierte en un nuevo thread, no posee referencias a mensajes previos.
- **NoReferenceShipmentView**: Envía mensajes simples, sin referencia.
- **ObjectReferenceShipmentView**: Envía mensajes que referencian algún objeto, no existiendo restricciones sobre la clase de este objeto.

La interfaz de estos tres componentes es la misma, ya que todos la heredan de la superclase abstracta, las subclases no agregan componentes visuales, solo agregan lógica.

Cada uno de estos componentes concretos fueron diseñados para formar interactuar con cada uno de los componentes *ShipmentManager* concretos, esto es, si se usa un determinado shipmentManager se debe utilizar un determinado ShipmentView:

ShipmentManager	Utilizar
NoReferenceShipmentManager	NoReferenceShipmentView
ThreadReferenceShipmentManager	ThreadReferenceShipmentView
ObjectReferenceShipmentManager	ObjectReferenceShipmentView



ShipmentView

FloorControlView

La principal función de este componente es informar sobre el desarrollo del protocolo de la aplicación, esto es, si este existe, mostrar el estado en que se encuentra el usuario actual.

Esta es una clase abstracta, de ella surgen cuatro componentes concretos. Los cuales se detallan a continuación.

Subclases

NoProtocolFloorControlView

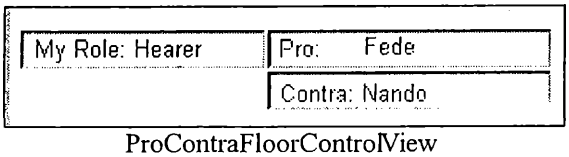
Este componente no muestra ninguna información, ya que no existe protocolo, la interfaz esta en blanco.

ProContraFloorControlView

Este componente muestra tres etiquetas, las cuales se detallan a continuación:

En el sector derecho muestra cual de los participantes del chat tiene el rol de *Pro* y cual el de *Contra*. En el sector izquierdo se muestra que rol posee el usuario local.

A medida que transcurre la conversación, el envío de mensajes al usuario le es permitido a determinados usuarios, como ya se dijo cuando se explico este protocolo, para proporcionar feedback al usuario este componente interactua con el *ShipmentView* habilitando y deshabilitando el botón de envío, indicando si el usuario puede o no realizar esta operación.



ProContraFloorControlView

DiscussionFloorControlView

Esta es una clase abstracta, de la misma surgen dos subclases, las que forman los componentes concretos.

Como ya explicamos anteriormente, existen dos tipos de componentes *DiscussionFloorControlManager*, por un lado el *AutoMediator* y por otro el *HumanMediator*.

Cuando se utiliza el *AutoMediator*, todos los usuarios del chat poseen el rol *DiscussionParticipantUserRole*, lo cual nos lleva a utilizar solo el componente *ParticipantDiscussionFloorControlView* como FloorControlView.

En el caso de utilizar *HumanMediator*, ahora tendremos varios usuarios cumpliendo el rol de *DiscussionParticipantUserRole*, y uno de *DiscussionMediatorUserRole*. Para los participantes, la interfaz el componente FloorControlView no cambia, se utiliza el mismo que se menciona en el caso anterior. La interfaz del usuario mediador, deberá utilizar como FloorControlView un componente *MediatorDiscussionFloorControlView*, ya que este provee la funcionalidad para que se lleve a cabo la discusión.

Para ello en el momento de mostrar la interfaz, esta se genera dinámicamente, dependiendo del rol de usuario se utilizara un determinado FloorControlView.

Ambos componentes se detallan a continuación:

ParticipantDiscussionFloorControlView

Este componente esta diseñado para interactuar con un *DiscusionFloorControlManager,* cualquiera de sus subclases, el único requerimiento es que el usuario local posea el rol de participante de la discusión. El mismo esta compuesto por un botón y una etiqueta. El botón, posee dos funciones, muestra el estado actual del usuario, feedback, y a su vez permite realizar operaciones para cambios de estado.

La etiqueta muestra quien es el participante que posee la palabra, esto emula la acción de ponerse de pie en un auditorio para hablar.

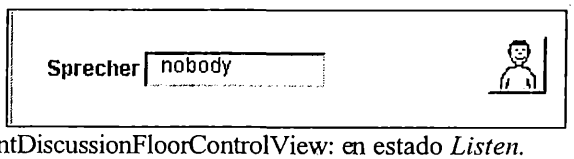
El *FloorControlView* interactua con el *ShipmentView,* para deshabilitar y habilitar el envío de mensajes.

Al iniciarse la aplicación el primer estado del participante es **Listen** el botón muestra al personaje de color amarillo, el envío de mensajes esta deshabilitado. Al presionar el botón, ejecuta la operación *request,* obteniendo como resultado el cambio de estado del usuario pasando al estado **WaitingToSpeech**.

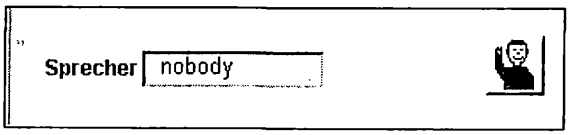
Ahora la imagen del botón muestra al personaje de rojo, con la mano levantada, emulando la situación real, el envío de mensajes se encuentra deshabilitado. La acción de presionar el botón ejecuta la operación *cancelRequest,* llevando al usuario al estado **Listen** nuevamente.

Cuando el mediador realiza la operación *setTurn* sobre un usuario este pasa de estado **WaitingToSpeech** a **AllowedToWrite,** en este nuevo estado el personaje se muestra de color verde y en la etiqueta se muestra el nombre del usuario local, habilitando el envío de mensajes. Al presionar el botón, se ejecuta la acción *cancelWrite,* pasando al estado **Listen**

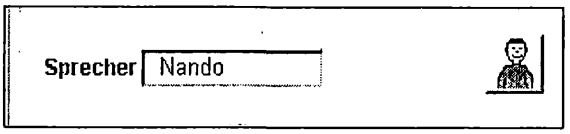
Ahora el usuario puede realizar dos acciones, por un lado enviar un mensaje y por otro cancelar el envío. Cualquiera de estas acciones generan el pase al estado inicial del usuario, este es **Listen**



ParticipantDiscussionFloorControlView: en estado *Listen.*



ParticipantDiscussionFloorControlView: en estado *WaitingToSpeech.*



ParticipantDiscussionFloorControlView: en estado *AlloweedToWrite.*

MediatorDiscussionFloorControlView

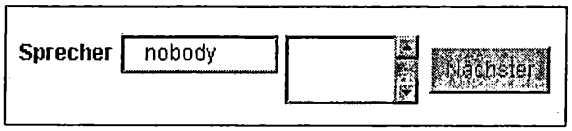
Este componente esta diseñado para utilizarse bajo el protocolo de discusión, imponiendo la restricción de que el rol del usuario local sea *DiscussionMediatorUserRole*.

La interfaz, descripta de izquierda a derecha, consta de una etiqueta, una lista y un botón. La etiqueta indica cual de todos los participantes es el que tiene la palabra para hablar. El contenido de la lista es la colección de participantes en estado *WaitingToSpeech*, la misma se completa automáticamente, ya que es un slot computado, y el botón a su derecha es el que realiza la operación de dar la palabra a un participante esperando su turno.

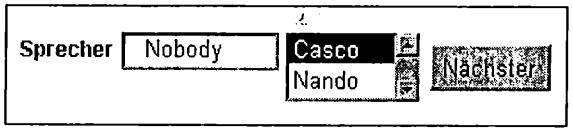
Inicialmente el mediador se encuentra en el estado **WaitingToGiveTurn**. En este estado el botón se encuentra deshabilitado, ya que no existen participantes esperando su turno o un de ellos posee la palabra.

El pase al siguiente estado, **GiveTurn**, se produce cuando ninguno de lo participantes tiene la palabra y existen participantes esperando su turno. En este caso la lista se encuentra llena y el botón se habilita. Presionando el botón se entrega el turno al participante seleccionado de la lista, se ejecuta la operación *setTurn*, lo que provoca el pase al estado inicial nuevamente.

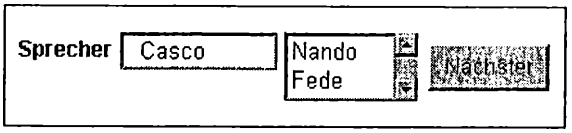
El mediador puede enviar mensajes en el momento que lo desee, no importando si la palabra la posee un participante o ninguno, por lo tanto el botón *send* se encuentra siempre habilitado.



Mediador en estado *WaitingToGiveTurn*



Mediador en estado GiveTurn



Mediador en estado *WaitingToGiveTurn*, con participantes esperando su turno.

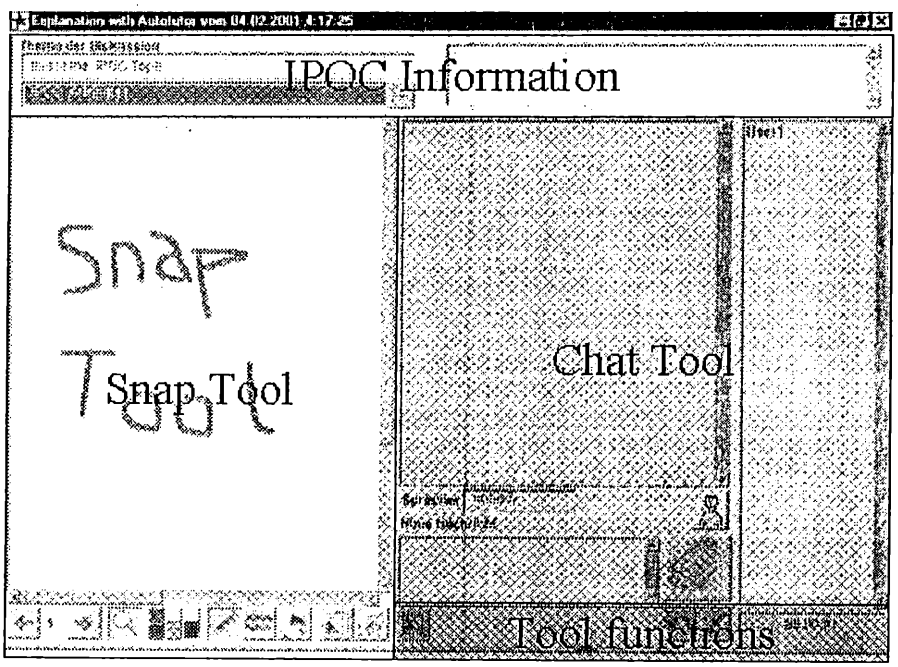
Cada uno de estos componentes esta diseñado para actuar como contrapartida local de un determinado FloorControlManager, los mismos se detallan a continuación:

FloorControlManager	Utilizar
ProContraFloorControlManager	ProContraFloorControlView
NoProtocolFloorControlManager	NoProtocolFloorControlView
AutoMediatorDiscussionFloorControlManager	MediatorDiscussionFloorControlView
HumanMediatorDiscussionFloorControlManager	MediatorDiscussionFloorControlView ParticipantDiscussionFloorControlView (*)

* en este caso se utilizan los dos componentes, la interfaz de usuario mostrará el correspondiente dependiendo del rol que posea el usuario.

Capítulo 7:
Aplicaciones creadas con Chatblocks

El ejecutable entregado contiene cinco herramientas, que forman parte del proyecto L3, La estructura de estas es en general la misma, consta de tres partes principales, por un lado la información del IPOC, parte superior de la interfaz, en el sector central izquierdo se encuentra el SNAP, que es una pizarra de dibujo cooperativa, en el sector central derecho un CHAT y por ultimo en el sector inferior derecho se hallan las demás funciones. Cada herramienta varía de las demás en el componente Chat que posee, estos chats son el resultado de Chatblocks.



Las herramientas chat

Simple Chat

Este chat forma parte de la herramienta **SnapChat**.

Este es un chat común. No posee un protocolo explícito, sus usuarios no asumen ningún tipo de rol, la visualización de los mensajes es en forma de lista ordenados en forma creciente por su fecha de creación.

La parte compartida, **SimpleChatSharedApplication** es subclase de **ChatSharedApplication**, utilizando los siguientes componentes:

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

En contraposición, la parte local de la aplicación, **SimpleChatLocalApplication,** es una subclase de **ChatLocalApplication**, utilizado los siguientes componentes:

- *UsersActivityView*
- *NoProtocolFloorControlView*
- *NoReferenceShipmentView*
- *ReceptionView*: donde el *viewer* es una instancia de **MessageViewerList** y *orderer* un **MessageOrdererByDate**

Thread Chat

Este chat forma parte de la herramienta **Synchronic Discussion**.

Este chat agrega semántica a los mensajes, los mensajes pueden contener una referencia a un mensaje enviado anteriormente, al cual llamaremos su "padre" o "antecesor".

Estas referencias forman una estructura jerárquica, formando threads o hilos de conversación.

Los mensajes sin referencia a un padre forman la cabeza del thread, esto quiere decir que introducen un subtema a la conversación, en cambio los que referencian a un padre están continuando un thread.

La conversación esta formada por varios hilos o threads, esto responde a un fenómeno en las conversaciones humanas. Las conversaciones rara vez se refieren a un tema en particular, por lo general, se tocan varios subtemas en una conversación, sin necesidad de existir relación entre estos, cada subtema forma un hilo de conversación.

La parte compartida, **ThreadChatSharedApplication** es subclase de **ChatSharedApplication**, utilizando los siguientes componentes:

- *ReceptionManager*
- *UsersManager*
- *NoRoleRoleManager*
- *NoProtocolFloorControlManager*
- *ThreadReferenceShipmentManager*

La parte local, **ThreadChatLocalApplication,** es una subclase de **ChatLocalApplication**, utilizado los siguientes componentes:

- *UsersActivityView*
- *NoProtocolFloorControlViewThreadReferenceShipmentView*
- *ReceptionView*: donde el *viewer* es una instancia de **MessageViewerTree** y *orderer* un **MessageOrdererByThread**

Human Mediator Discussion Chat

Este chat forma parte de la herramienta **Explanation**.

Este chat posee características que lo diferencian de las herramientas chat convencionales, ya que posee un protocolo explícito, la aplicación fuerza a los usuario a cumplir con este. Los usuarios cumplen un rol, en este caso habrá un *mediador* y varios *participantes*, el mediador es encargado de dirigir la conversación, los participantes deberán solicitar la palabra y esperar su turno para "hablar", el encargado de decidir a quien se le otorgará el turno es el mediador.

Los mensajes poseen referencias a otros mensajes, forman hilos o threads de conversación, al igual que lo hace **ThreadChat**.

La parte compartida, **HumanMediatorDiscussionChatSharedApplication** es subclase de **ChatSharedApplication**, utilizando los siguientes componentes:

- *ReceptionManager*
- *UsersManager*
- *DiscussionRoleManager*
- *HumanMediatorDiscussionFloorControlManager*
- *ThreadReferenceShipmentManager*

La parte local, **HumanMediatorDiscussionChatLocalApplication,** es una subclase de **ChatLocalApplication**, utilizado los siguientes componentes:

- *UsersActivityView*
- *MediatorDiscussionFloorControlView–ParticipantDiscussionFloorControlView*: La interfaz mostrará uno u otro según el rol que posea el usuario local.
- *ThreadReferenceShipmentView*
- *ReceptionView*: donde el *viewer* es una instancia de **MessageViewerTree** y *orderer* un **MessageOrdererByThread**

Auto Mediator Discussion Chat

Este chat forma parte de la herramienta **Explanation with Autotutor**.

Es similar al Human Mediator Discussion Chat, con la única diferencia que no existe mediador, ya que ahora el rol de medidador lo asume la aplicación. Todos los usuarios son particpantes de la discusión.

La parte compartida, **AutoMediatorDiscussionChatSharedApplication** es subclase de **ChatSharedApplication**, utilizando los siguientes componentes:

- *ReceptionManager*
- *UsersManager*
- *DiscussionRoleManager*
- *AutoMediatorDiscussionFloorControlManager*
- *ThreadReferenceShipmentManager*

La parte local, **AutoMediatorDiscussionChatLocalApplication,** es una subclase de **ChatLocalApplication**, utilizado los siguientes componentes:

- *UsersActivityView*
- *ParticipantDiscussionFloorControlViewThreadReferenceShipmentView*
- *ReceptionView*: donde el *viewer* es una instancia de **MessageViewerTree** y *orderer* un **MessageOrdererByThread**

Pro Contra Chat

Este chat forma parte de la herramienta **Pro / Contra**.

La principal innovación en este chat reside en el floor control. Aquí existe un protocolo al que la aplicación fuerza cumplir. Esta aplicación intenta emular situaciones de debate sobre una determinada idea, con dos participantes, uno a favor y otro en contra.

Los usuarios pueden asumir el rol de *Pro*, *Contra* o *Hearer*, la aplicación se encarga de alternar los tiempos de exposición a los participantes del debate.

La parte compartida, **ProContraChatSharedApplication** es subclase de **ChatSharedApplication**, utilizando los siguientes componentes:

- *ReceptionManager*
- *UsersManager*
- ProContra*RoleManager*
- *ProContraFloorControlManager*
- *NoReferenceShipmentManager*

En contraposición, la parte local de la aplicación, **ProContraChatLocalApplication,** es una subclase de **ChatLocalApplication**, utilizado los siguientes componentes:

- *UsersActivityView*
- *ProContrlFloorControlView*
- *NoReferenceShipmentView*
- *ReceptionView*: donde el *viewer* es una instancia de **MessageViewerList** y *orderer* un **MessageOrdererByDate**

Capítulo 8
Conclusiones

En el capitulo 5 se enumeraron la una serie de resultados que se esperaban obtener con este framework, las mismas están analizadas aquí:

- **Facilidad**: Con Chatblocks el programador de la aplicación no debe conocer internamente la estructura del framework, solo conocer la funcionalidad de cada uno de los componentes provistos por el mismo.

- **Rapidez** La construcción de una herramienta chat consiste en seleccionar que componentes se desean usar, o ser crear instancias de los componentes y agregarlos al chat, como lo hace un programador de GUIs cuando, selecciona que widgets necesita utilizar, crea sus instancias y las agrega a su GUI.

- **Flexibilidad**: Aporta mecanismos para la experimentación y evaluación de variaciones de herramientas chat. Ejemplo, protocolos explícitos, roles de usuario, formas de visualización, representaciones de usuarios, etc. Todos estos aspectos se pueden combinar, obteniendo un gran numero de chats posibles, sin necesidad de extender el framework.

- **Extensionalidad**: El framework permite la creación de nuevos componentes, por ser un framework de caja blanca, esto se obtiene mediante la subclasificación de los componentes provistos por Chatblocks. Logrando así poder modelar mas protocolos, roles, diferentes interfaces de usuario, etc.

- **Soporte para la implementación de protocolos explícitos**: El prototipo ha comprobado que Chatblocks permite implementar un protocolo explicito, el framework provee dos protocolos *Discussion* y *ProContra*.

Como se cubren los aspectos propuestos

Almacenamiento

El MessagesPool es una colección de mensajes, por ser un objeto mantenido por COAST, la misma no se destruye. Por lo tanto el almacenamiento es permanente.

Desarrollo de la conversación

Dado que COAST provee un mecanismo de permanencia de objetos y replicación, desde el punto de vista del programador no se distinguen entre estas dos situaciones, los mensajes que se envían siempre se almacenan en el **MessagesPool**, y los que se reciben se toman de esta colección. Por esa razón el framework provee soporte para la creación de chat Sincrónicos o Asincrónicos.

Número de Participantes

Permite que haya dos o más participantes del chat. En caso de necesitar limitar la cantidad de participantes a un mínimo o máximo, se puede configurar el mismo mediante los métodos **minParticipants** y **maxParticipants** que posee el **UsersManager**. Esto permite la flexibilidad a la hora de crear chats con restricciones al nivel de la cantidad de participantes.

Awareness

El **UsersManager** es el encargado de almacenar la información de awareness, éste sabe quien participa del chat y quien está conectado a la aplicación. El **UsersActivityView** es el encargado de mostrar al usuario que sucede con los demás participantes del chat, para ese fin, este posee una lista de todos los participantes del chat, de los cuales los usuarios conectados se dibujan con una fuente bold, mientras que los desconectados con fuente normal. La lista de participantes provee Awareness de Participación, mientras que el tipo de font en los participantes cubre el Awareness de Conexión.

Representación del autor del mensaje

El **ReceptionView** posee un viewer, este puede ser una lista o un árbol, los mismos muestran mensajes, cada mensaje es un objeto compartido (CoastModel), por lo tanto este objeto posee es el modelo de un MessageApplication, que es la capa de aplicación, cada MessageApplication implementa un método para representar un mensaje. Esta clase es abstracta, y para ello existen tres subclases concretas del mismo, cada una de ellas visualiza los mensajes de una forma diferente:

- Nombre: MessageNameApplication
- Icono: MessageIconApplication
- Fonts: MessageFontColorApplication

Clasificación de Usuarios

El RoleManager es el encargado de cubrir este aspecto. El mismo asigna un UserRole a cada ChatUser, como RoleManager es una clase abstracta a la hora de crear una aplicación se deberá se debe instanciar alguna de sus subclases. El framework cubre este aspecto de la siguiente forma:

- Sin Roles: Para obtener esta modalidad, se debe usar como RoleManager el componente NoRoleRoleManager, el cual asigna un rol NoRole al usuario.
- Con Roles: Existen dos RoleManager implementados que cubren este aspecto, uno es el ProContraRoleManager y el otro es el DiscussionRoleManager.

Floor Control

El responsable de cubrir este aspecto es el FloorControlManager. Para ocupar este lugar, Chatblocks provee cuatro componentes concretos, cada uno de ellos tiene la finalidad de suministrar diferentes variaciones de este aspecto:

- Sin Protocolo: Usando el componente **NoProtocolFloorControlManager**.
- Con Protocolo: Para obtener esta variante se pueden usar cualquiera de los tres componenetes que se detallan a continuación:
  - **ProContraFloorControlManager:** Representa un debate sobre un tema dado entre dos personas, un Pro y un Contra, y varios oyentes, que conforman el publico.
  - **AutoMediatorDiscussionFloorControlManager:** Representa una discusión entre varias personas, por ejemplo una conferencia de prensa o una situación de participación de alumnos en una clase, la misma consta de participantes y un mediador, que es el encargado de dar la palabra, de a uno a la vez, a los participantes que la soliciten. En este caso el

mediador en automático y posee una estructura FIFO para asignación de turnos.

- **HumanMediatorDiscussionFloorControlManager:** Idem a la situación anterior, pero con la diferencia que uno de los usuarios del chat asume el rol de mediador.

Semántica de los mensjes

Para cubrir este aspecto Shipment Manager existen tres variaciones de Shipment Manager, cada uno de estos componentes aporta una modalidad de este aspecto:

- **NoReferenceShipmentManager:** Este componente cubre la modalidad de envío de mensajes sin semántica.
- **ThreadReferenceShipmentManager:** Utilizando este componente como ShipmentManager se obtienen hilos o threads de conversación. Es decir mensajes que responden a otros.
- **ObjectReferenceShipmentManager:** Este componente aporte al chat mensajes capaces de referenciar algún objeto, en este caso no existen restricciones de tipos de objetos.

Formas de Recepción

Este aspecto es responsabilidad del ReceptionManager, cada usuario recibe todos los mensajes que se han enviado a lo largo de la conversación, tanto los enviados mientras estaba OnLine, como los enviados mientras no lo estaba.

Filtros

EL filtrado de información es de suma utilidad cuando lidiamos con grandes cantidades de la misma. El ReceiverManager provee tres modalidades para cubrir este aspecto. Utilizando como MessageFilter el subcomponente indicado para cada caso en particular:

- **Autor**: subcomponente: **MessageFilterByAuthor.**
  Método para la selección: **setFilterbyAuthor.**

- **Fecha**: subcomponente: **MessageFilterByDate**
  Método para la selección: **setFilterbyDate.**

- **Thread**: subcomponente: **MessageFilterByThread**
  Método para la selección: **setFilterbyThread.**

Receptor

Los mensajes poseen un emisor, que es su autor, pero no especifican un receptor, esto es así porque con Chatbloks se desea crear aplicaciones donde cada mensaje le es enviado a cada usuario del chat, por lo tanto todos los participantes se convierten en receptor. El ReceiverManager es en responsable de cubrir este aspecto.

Criterio de Ordenación

Para la cobertura de este aspecto, el framework posibilita la ordenación de los mensajes de tres maneras. De esta función es responsable **ReceptionView**, mas precisamente el subcomponente *orderer* de éste. Para seleccionar un criterio de ordenación de mensajes basta con invocar un método determinado del **ReceptionView.**

Los criterios de ordenación que soporta el framework son:

- **Por Autor:** invocando el método *orderByAuthor* del componente **ReceptionView.**
- **Por Thread:** invocando el método *orderByThread* del componente **ReceptionView.**
- **Por Fecha**: invocando el método *orderByDate* del componente **ReceptionView.**

Visualización de los mensajes

El responsable de cubrir este aspecto es el componente **ReceptionView,** el mismo permite la visualización de dos maneras posibles:

- **Lista:** invocando el método view*AsList* del componente **ReceptionView.**
- **Arbol:** invocando el método *viewAsTree* del componente **ReceptionView.**

Formato de envío

El framework cubre una sola variación de este aspecto, esta es **texto plano,** ya que el **ShipmentView** no provee funcionalidad para asignar estilos y tipos de fuentes a los mensajes.

Unidad de envío

En este caso, Chatblocks adopta como unidad de envío el *mensaje*. Esto es que los aportes de los participantes del chat se escriben en la maquina local y luego son enviados a los demás usuarios.

Conclusiones

La creación del prototipo demostró que el framework facilita la implementación de las herramientas chat, ya que desarrollar un chat con el framework lleva unos pocos minutos, el programador solo tiene que agregar componentes al chat.

La flexibilidad obtenida y la capacidad para soportar protocolos explícitos, en las aplicaciones finales, serán de gran utilidad a la hora de experimentar con nuevas herramientas.

Chatblock ha demostrado ser capaz de crear herramientas chat innovadoras, utilizando tecnología de punta.

Future Work

Una vez completado el prototipo Smalltalk, se realizará la primera versión estable del framework en Java, el mismo estará realizado sobre un framework cooperativo con características similares al COAST, llamado DyCE.

Permitir el envió de mensajes con tipo, enriqueciendo la semántica de los mensajes.

Enriquecer la interfaz de usuario mediante la realización de widgets específicos para la versión Java.

Extensión del modelo para soportar estadísticas sobre los mensajes enviados y recibidos por los usuarios.

Agregar mecanismos de monitoreo de actividad del usuario, para detectar situaciones en las que la aplicación se encuentra activa pero el usuario no se encuentra observándola. Ej. detectar si el usuario esta en estado **"Away"**.

Otra de las futuras modificaciones que se desean realizar como future work es transformar a Chatblocks en un framework de caja negra.

Capítulo 9
Bibliografía

1. R. Johnson, B. Foote. Designing Reusable Classes. OOPSLA'91.

2. M. Fayad, D. Schmidt. "Object Oriented Application Frameworks". Communications of the ACM Octubre 1997/Vol. 40, No 10, page 32.

3. R. Johnson. Frameworks = Componets + Patterns. Communications of the ACM Octubre 1997/Vol. 40, No 10, page 39.

4. D. Vronay, M. Smith, S. Drucker. Alternative Interfaces for Chat. CHI'99.

5. F. Viegas, J. Donath. Chat Circles. CHI'99.

6. F. Viegas, J. Donath, K. Karahalios. "Visualizing Conversation". MIT Media Lab. 1999.

7. J. Hewitt. Beyond Threaded Discourse. WebNet'97.

8. E. Gamma, R. Helm, R. Johnson, J. Vessides. Design Patterns: Elements for reusable Object-Oriented Programming. Addison-Wesley, 1995.

9. Douglas C. Schmidt, Mohamed E. Fayad, Ralf E. Johnson. Building Application Frameworks. Wiley1999.

10. 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.

11. Campbell, R.H. and Islam, N. A technique for documenting the framework of an object-oriented system. Computing Systems

12. Fayad, M.E. and Hamu, D.S. Object-oriented enterprise frameworks: Make vs. buy decisions and guidelines for selection. Commun. ACM, submitted for publication.

13. Fayad, M.E. and Hamu, D.S. Object-Oriented Enterprise Frameworks. Wiley, NY, 1997.

14. Gamma, E, Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Software Architecture. Addison-Wesley, 1995.

15. Hueni, H., Johnson, R., and Engel, R. A framework for network protocol software. In Proceedings of OOPSLA'95, (Austin, Texas, Oct.1995).

16. Pree, W. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1994.

17. 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.

18. Fayad, M.E., Schmidt, D.C., and Johnson, R.E. Object-Oriented Application Frameworks: Problems and Perspectives. Wiley, NY, 1997.

19. 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

20. Fayad, M.E., Schmidt, D.C., and Johnson, R.E. Object-Oriented Application Frameworks: Implementation and Experience. Wiley,1997.

21. Ellis, C., Gibbs, S., and Rein, G., Groupware: Some Issues and Experiences, Communications of the ACM, Vol. 34, No. 1, 1991, 39-58

22. Ackerman, M., Starr, B., Social Activity Indicators for Groupware, Computer, June 1996, 37-42

23. Altun, A., Interaction Management Strategies on IRC and Virtual Chat Rooms, SITE 98 proceedings, March 10-14, 1998, 1223-1227

24. Beaudouin-Lafon, M., Karsenty, A., Transparency and Awareness in Real-Time Groupware Systems, UIST proceedings, 1992, 171-180

25. Dourish, P., Bellotti, V., Awareness and Coordination in Shared Workspaces, CSCW proceedings, 1992, 107-114

26. Dourish, P., Bly, S., Portholes: Supporting Awareness in a Distributed Work Group, CHI proceedings, 1992, 541-547

27. COAST: http://www.opencoast.org

28. ICQ: http:// www.icq.com

29. mIRC: http://mirc.stealth.net

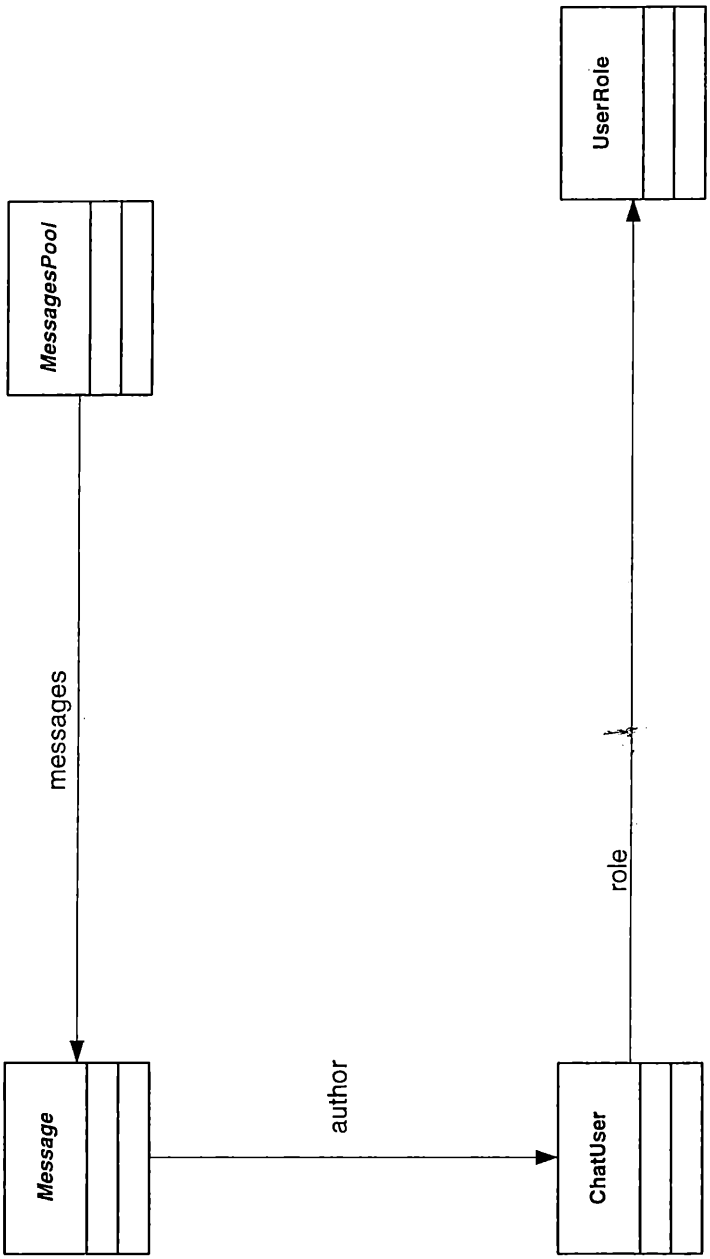
30. MS-Chat: http://www.microsoft.com

31. Yahoo Messager: http://www.yahoo.com

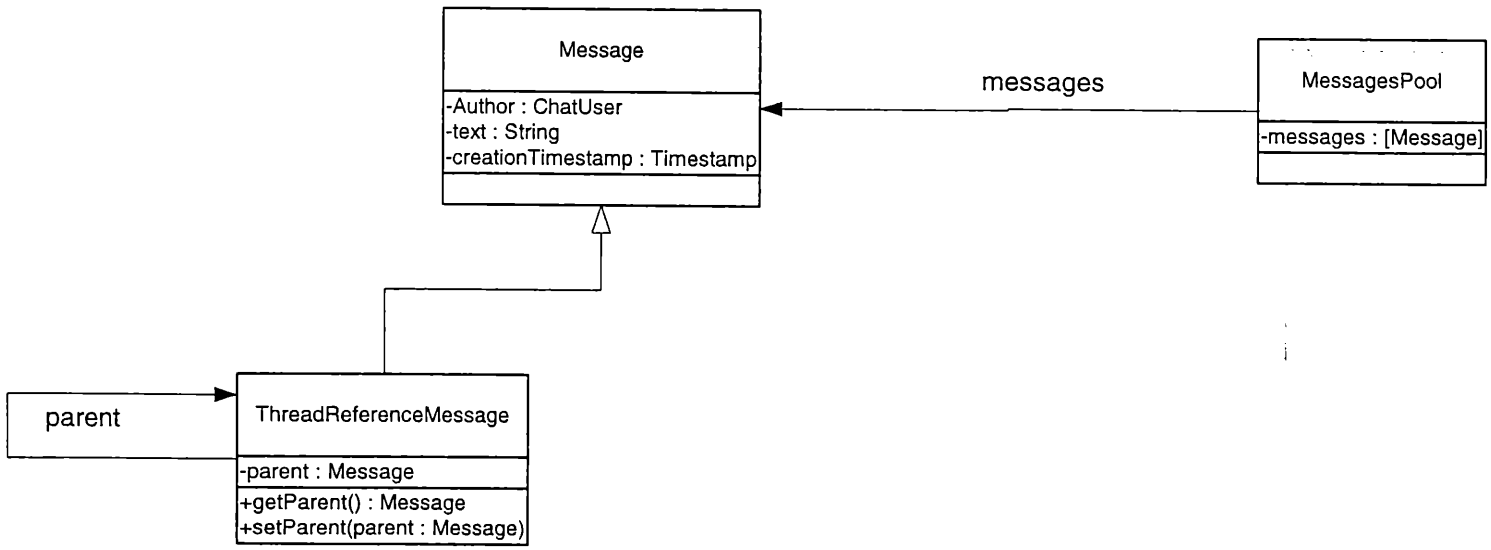
Apéndice A

Diagramas de clase del Framework

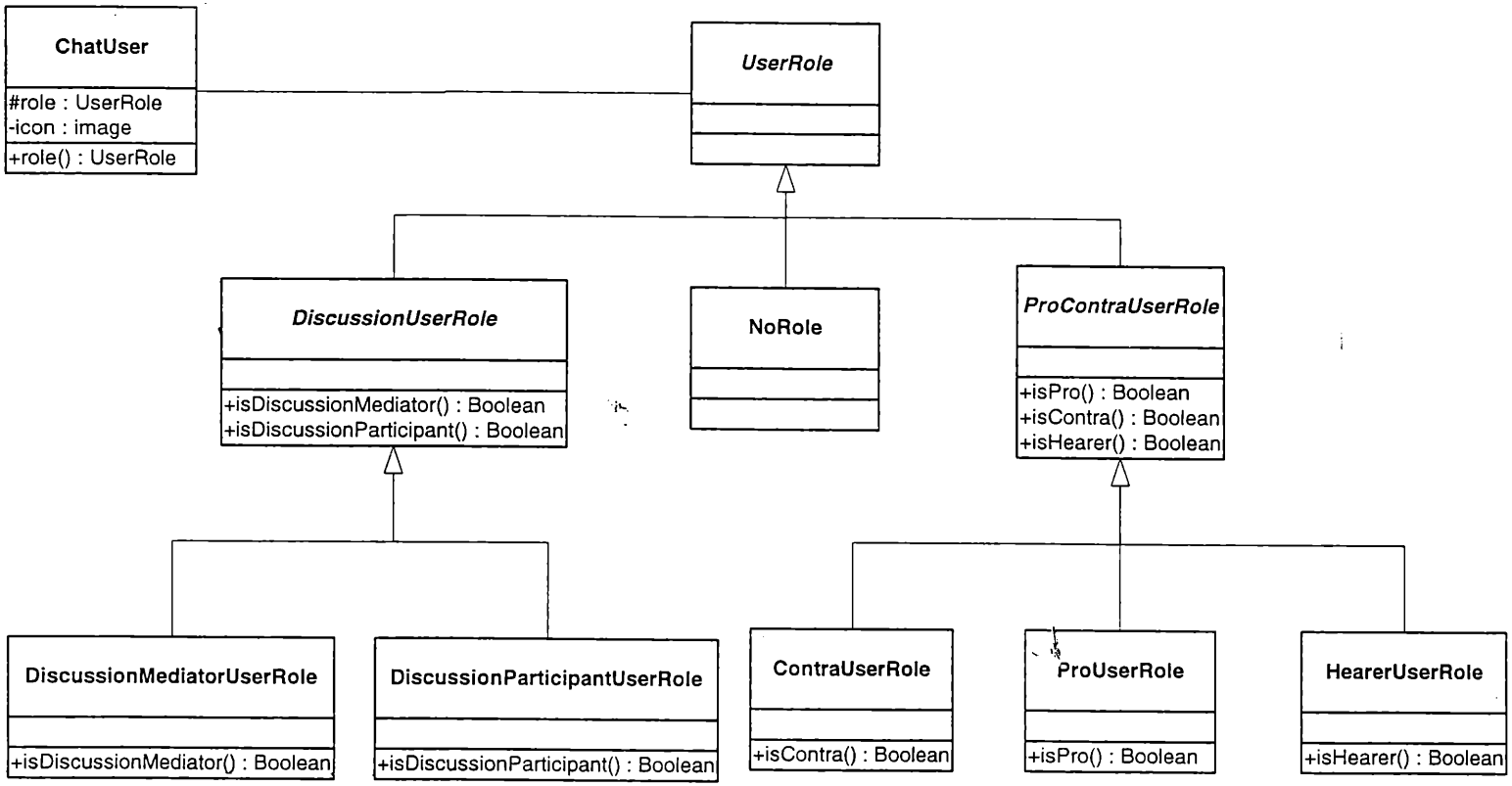
Domain Model



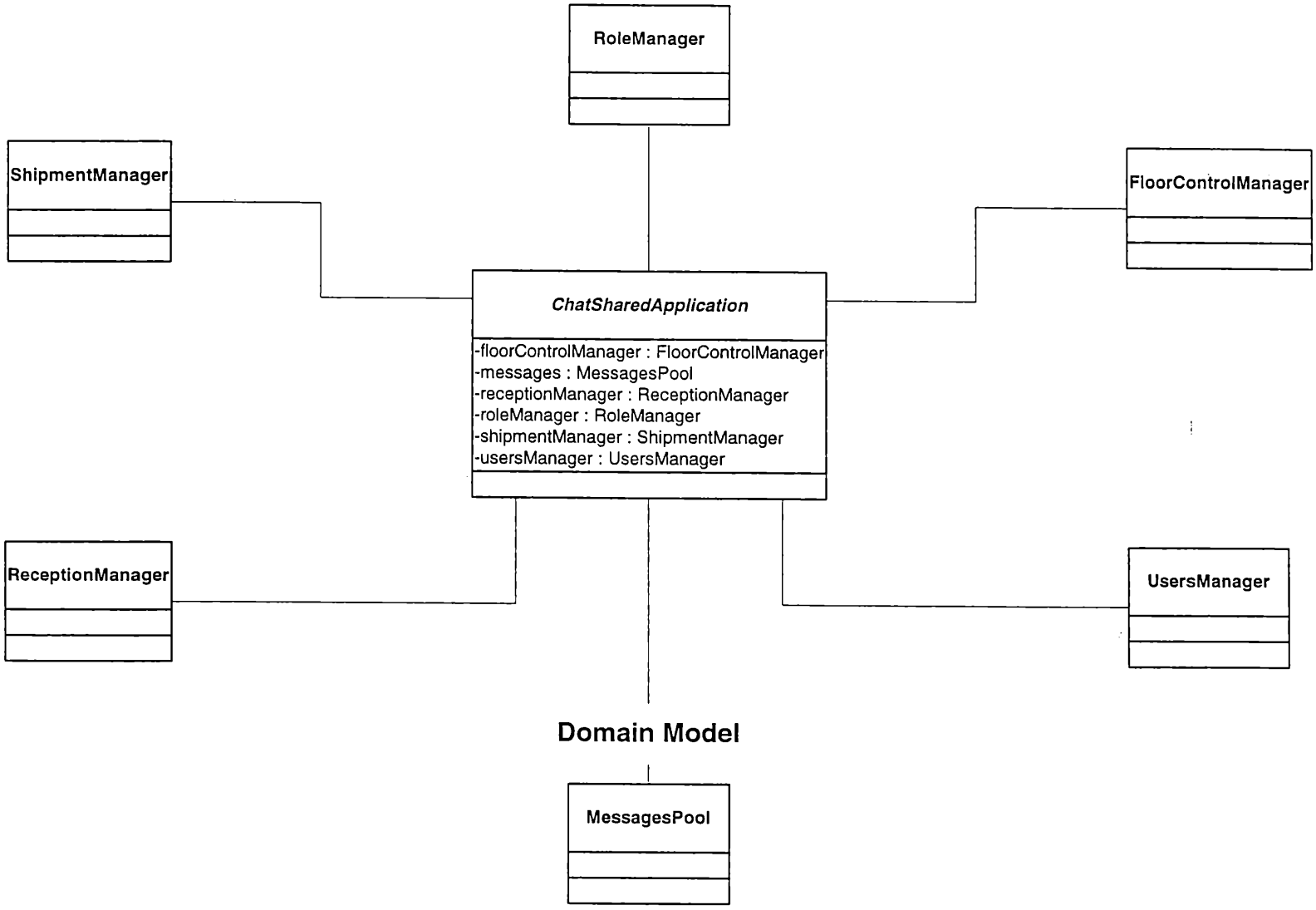
MessagesPool - Message



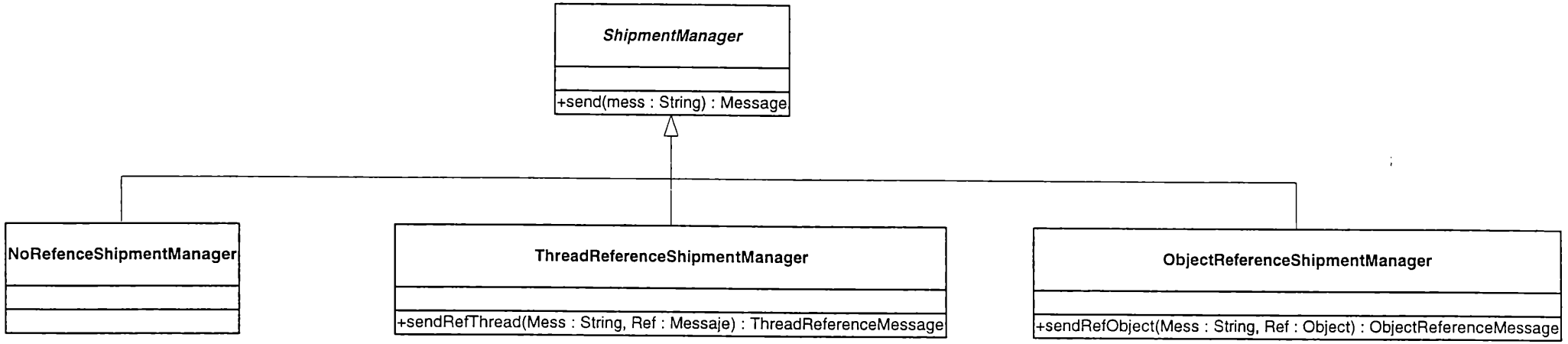
UserRole


ChatUser
#role : UserRole
-icon : image
+role() : UserRole
UserRole
DiscussionUserRole
+isDiscussionMediator() : Boolean
+isDiscussionParticipant() : Boolean
NoRole
ProContraUserRole
+isPro() : Boolean
+isContra() : Boolean
+isHearer() : Boolean
DiscussionMediatorUserRole
+isDiscussionMediator() : Boolean
DiscussionParticipantUserRole
+isDiscussionParticipant() : Boolean
ContraUserRole
+isContra() : Boolean
ProUserRole
+isPro() : Boolean
HearerUserRole
+isHearer() : Boolean

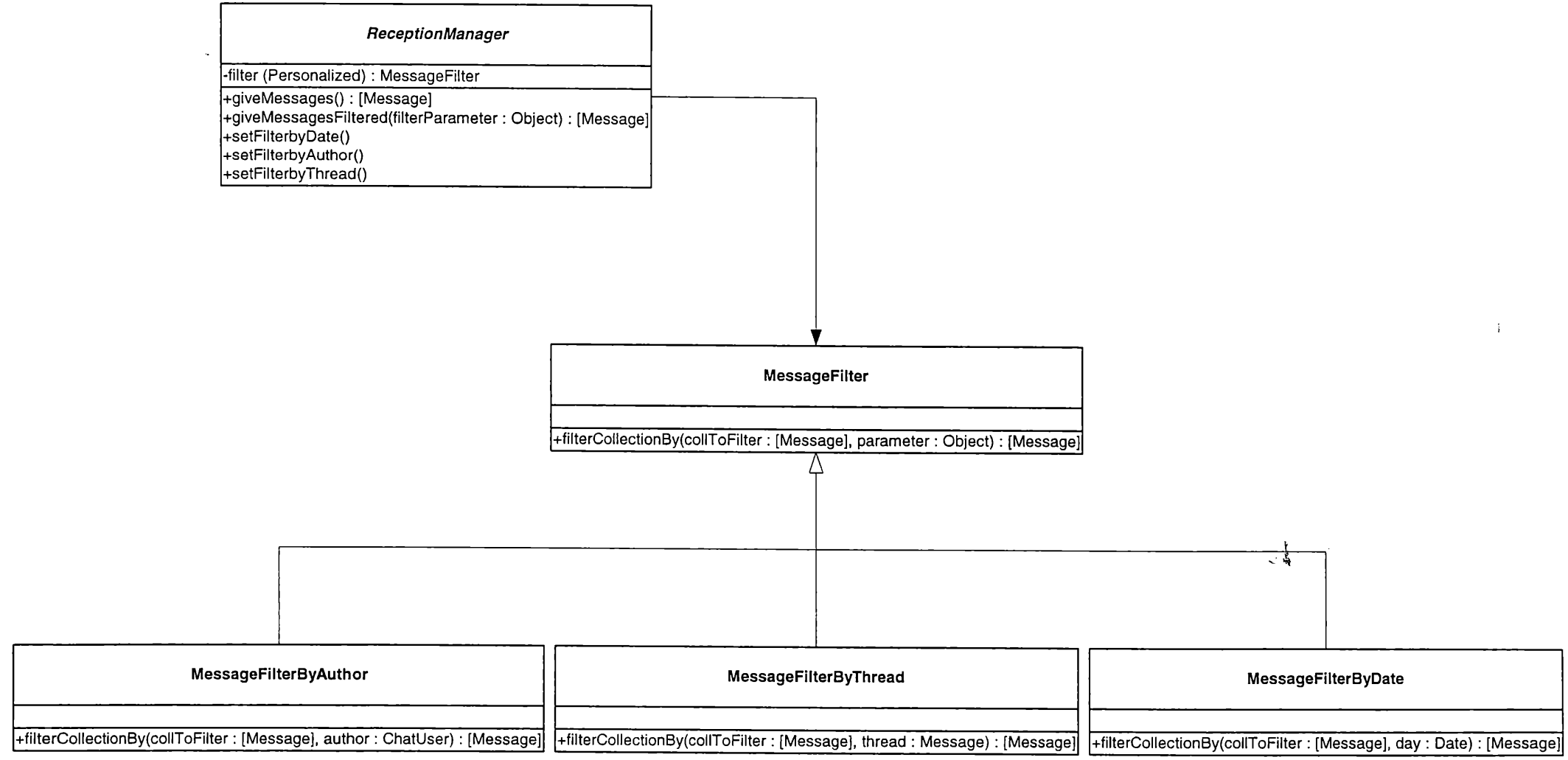
ChatSharedApplication



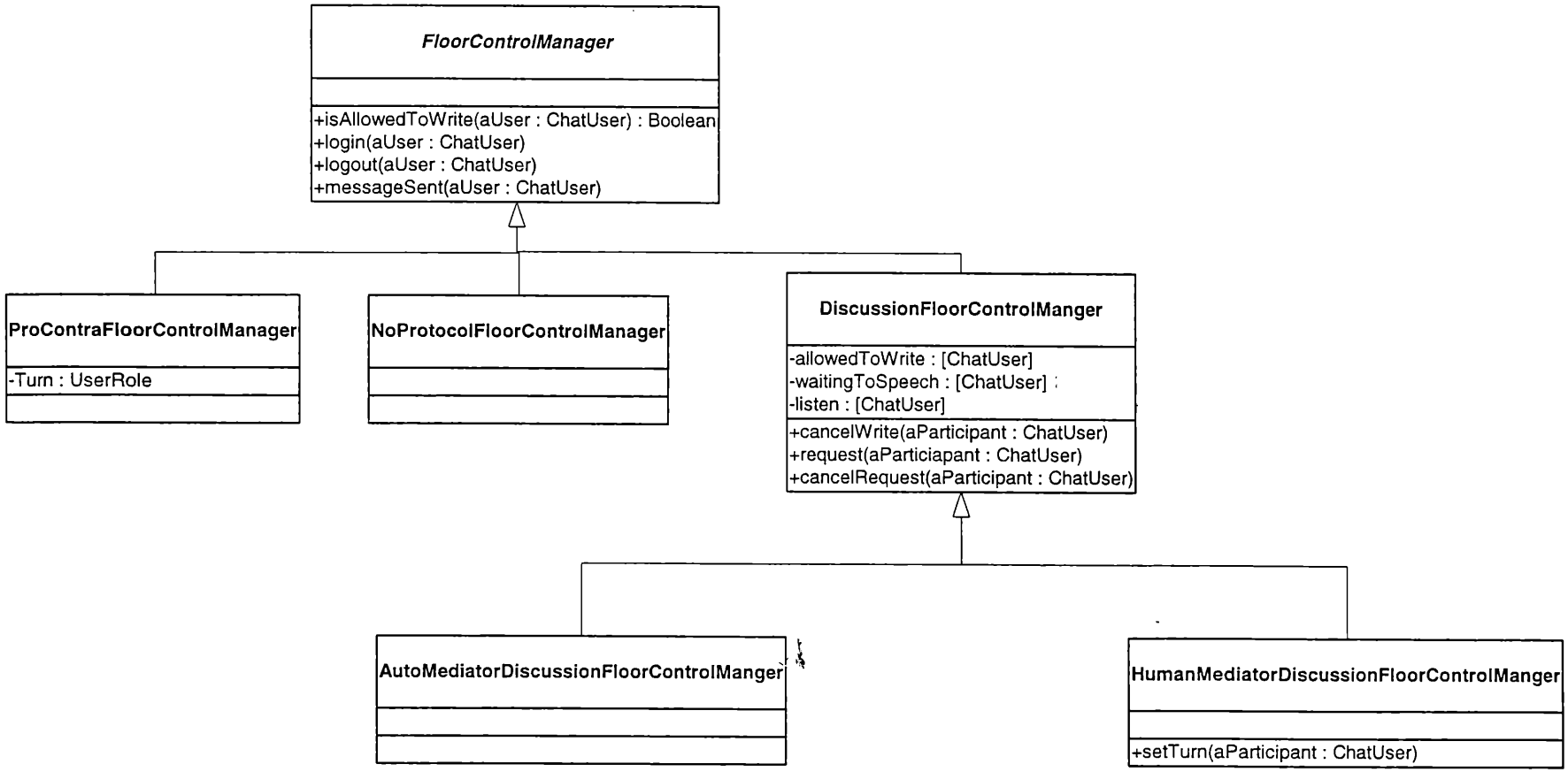

ShipmentManager


ShipmentManager
+send(mess : String) : Message
NoRefenceShipmentManager
ThreadReferenceShipmentManager
+sendRefThread(Mess : String, Ref : Messaje) : ThreadReferenceMessage
ObjectReferenceShipmentManager
+sendRefObject(Mess : String, Ref : Object) : ObjectReferenceMessage

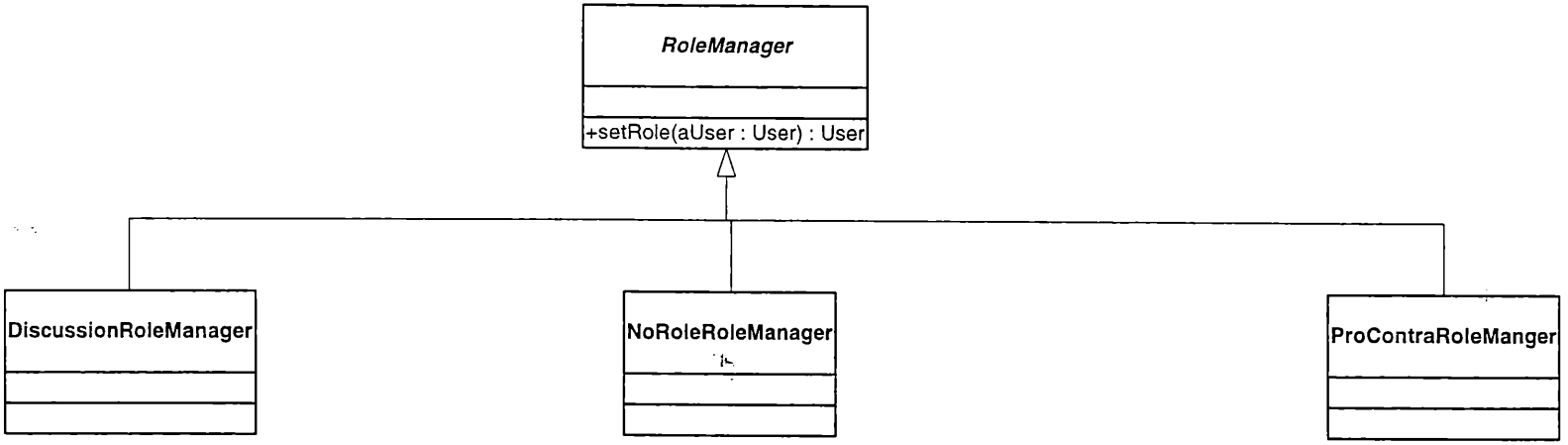
ReceptionManager



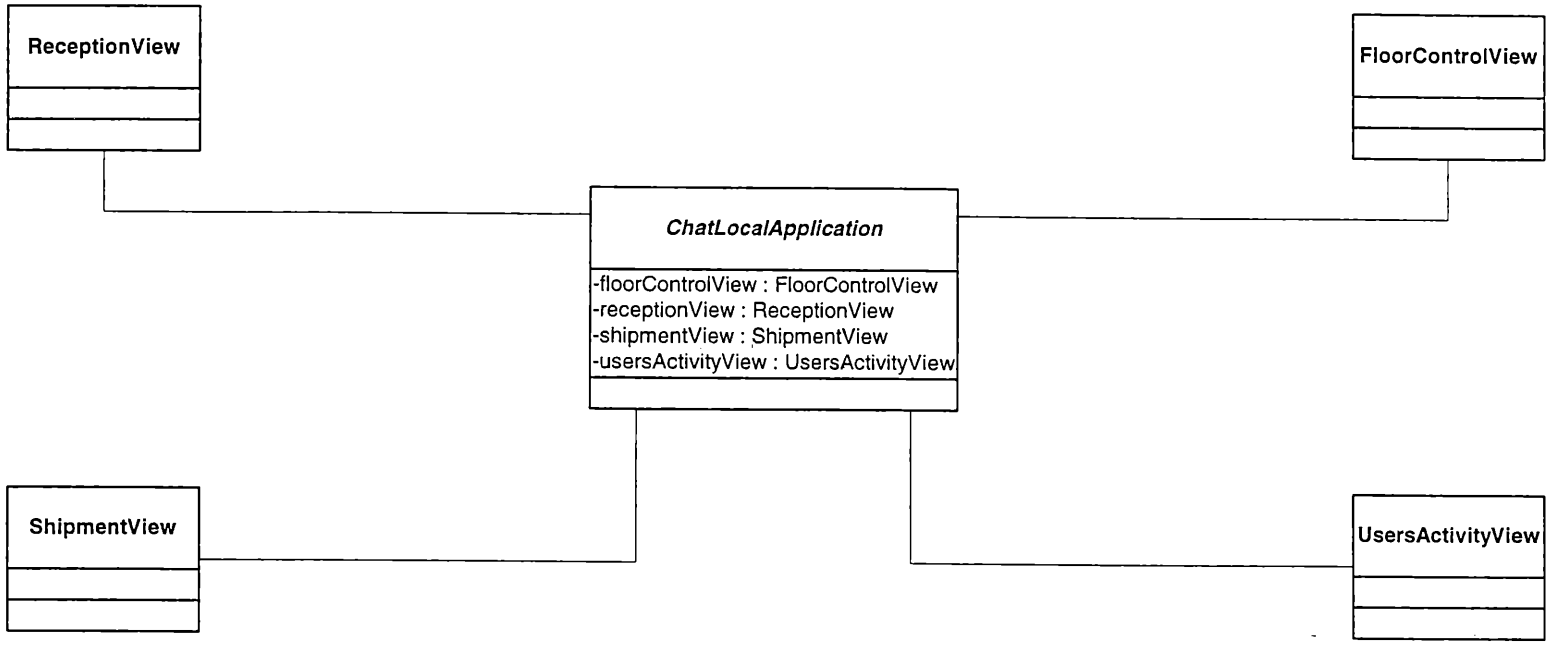
FloorControlManager



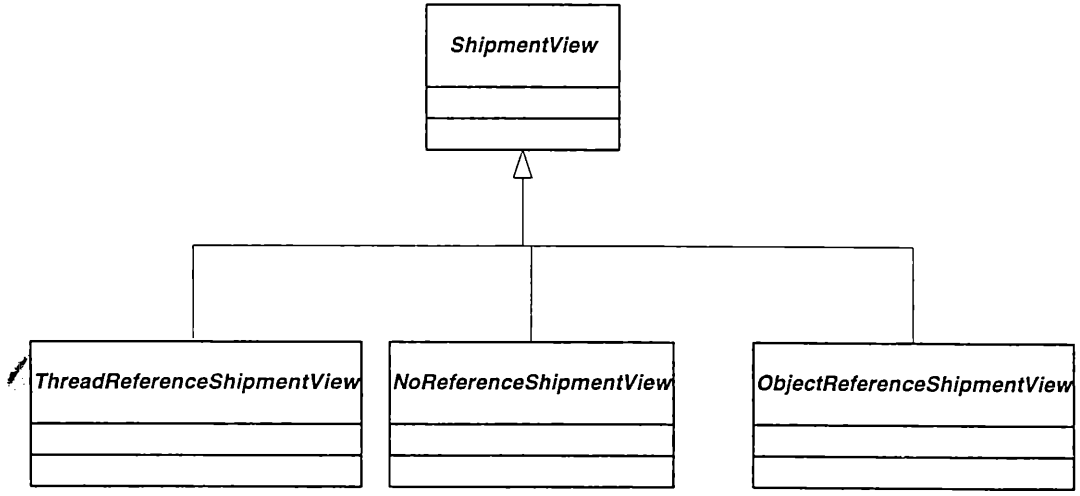
RoleManager



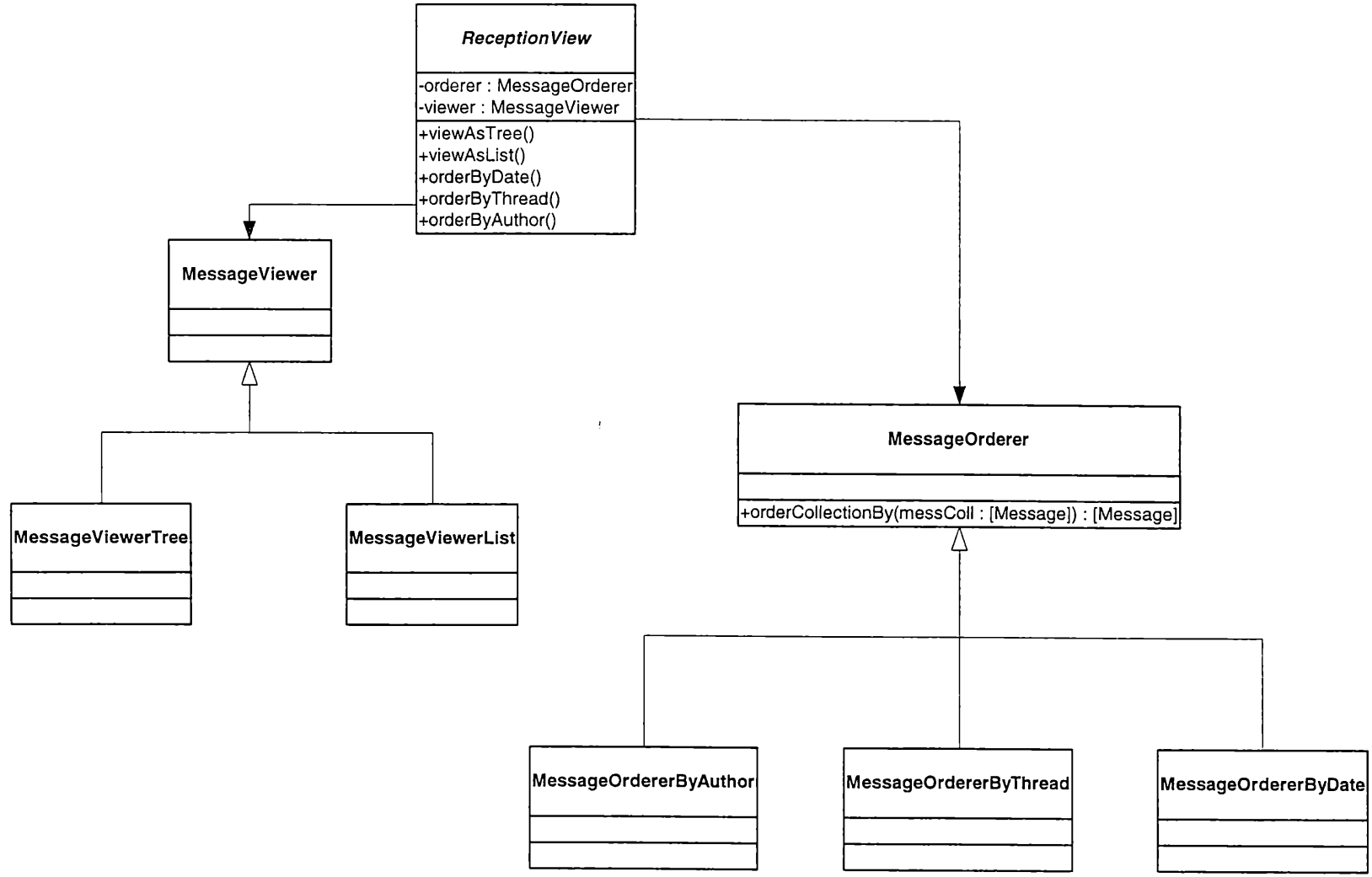
ChatLocalApplication



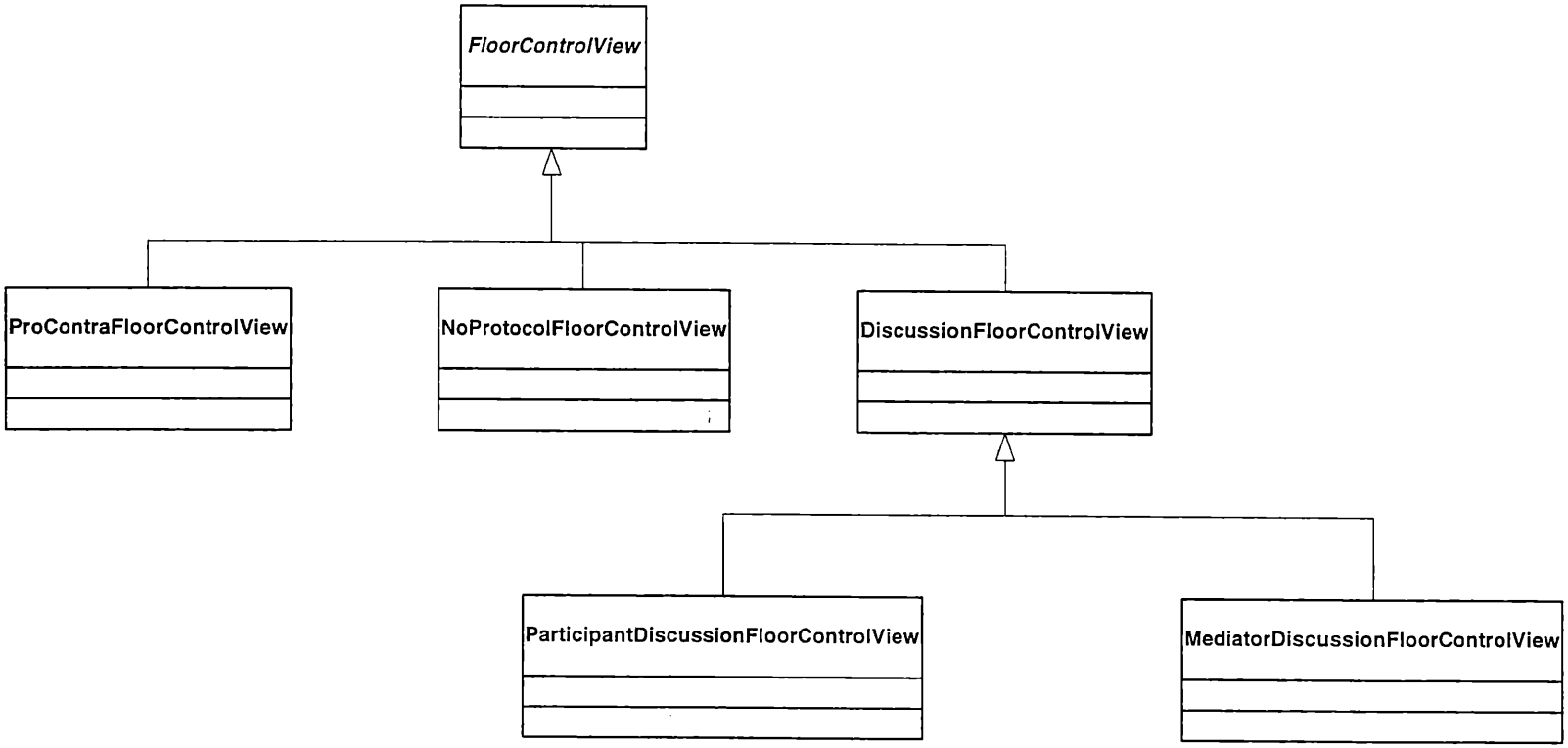
ShipmentView



ReceptionView



FloorControlView

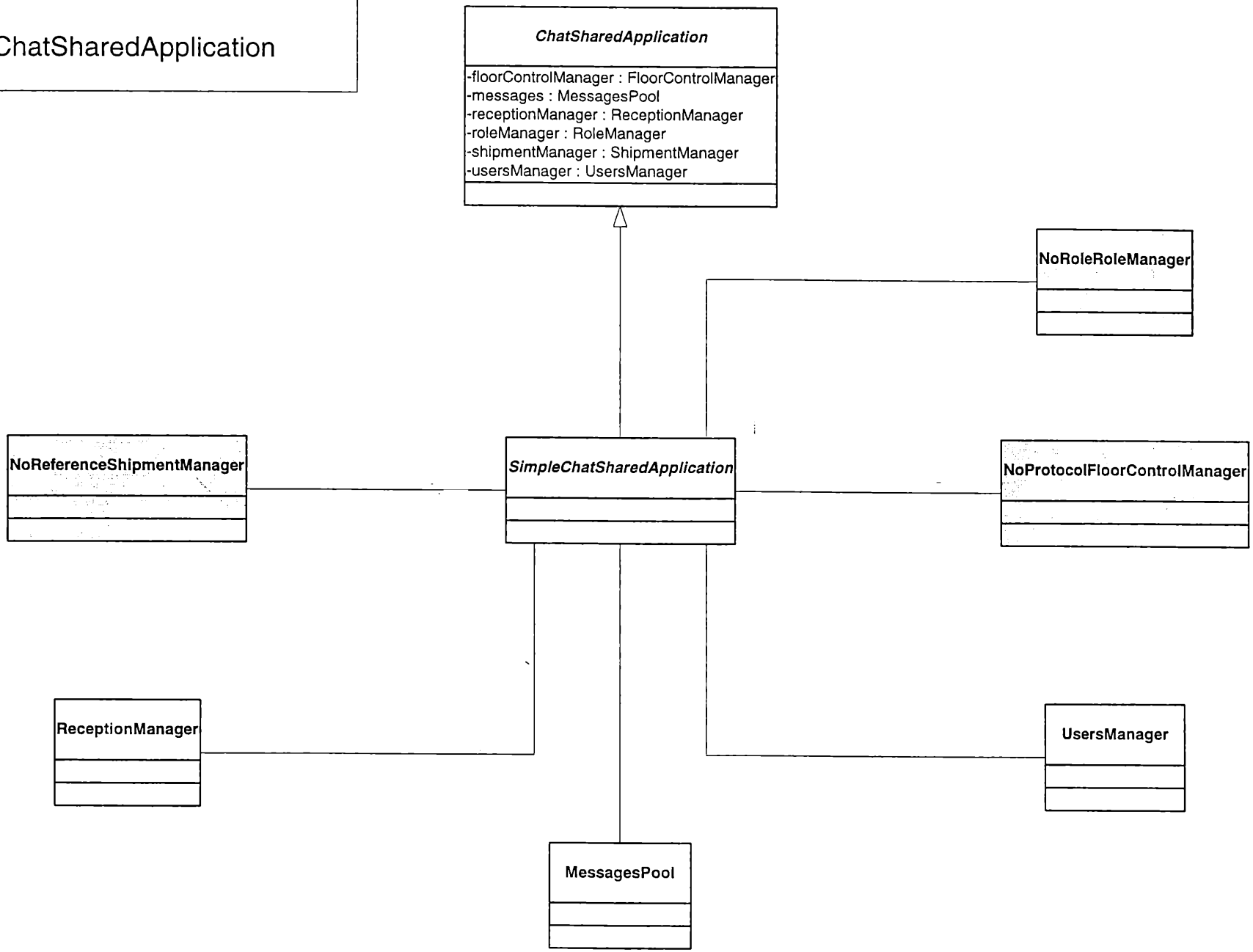


Apéndice B

Diagramas de clase de las aplicaciones creadas con Chatblocks

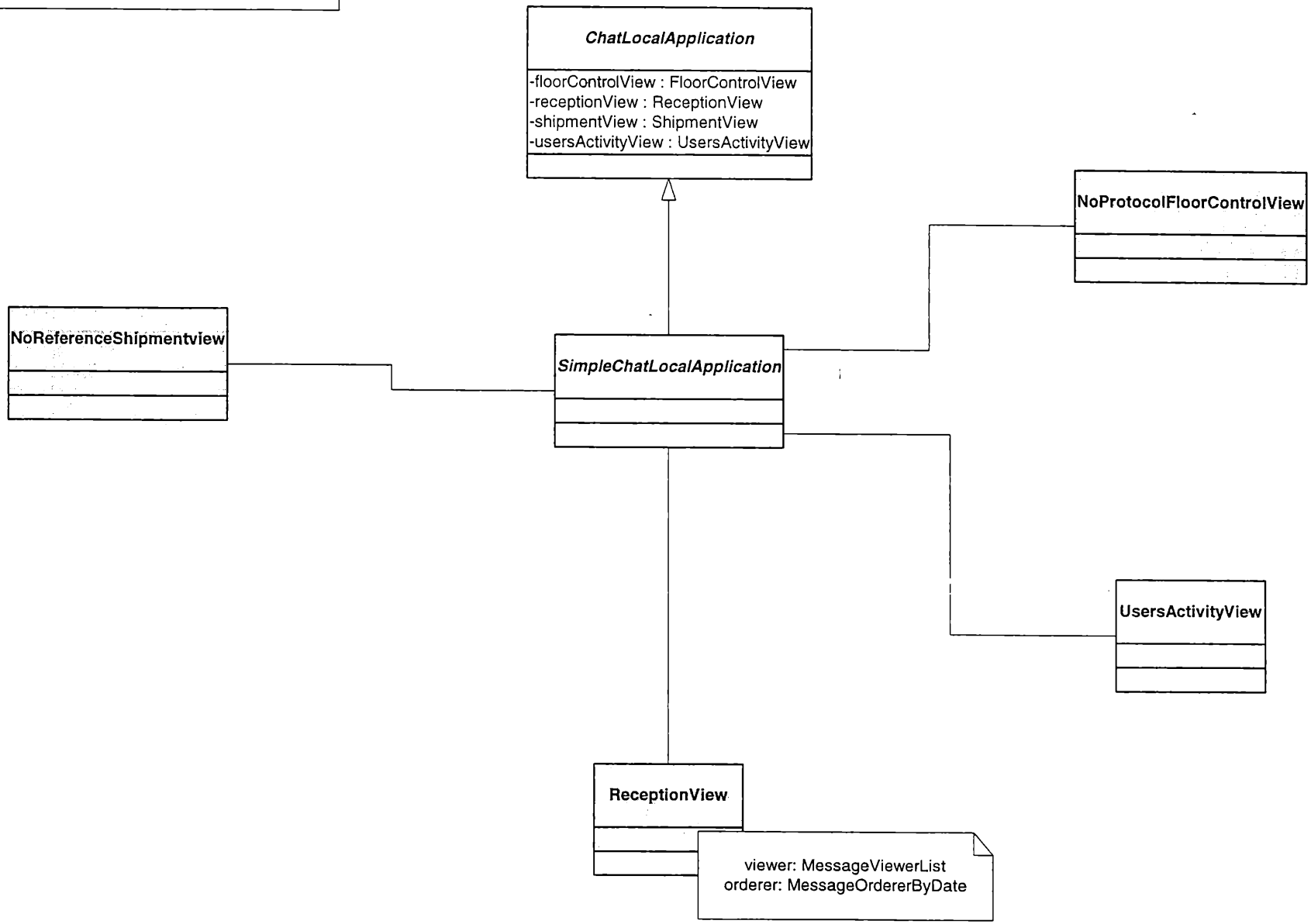
Snap Chat

SimpleChatSharedApplication



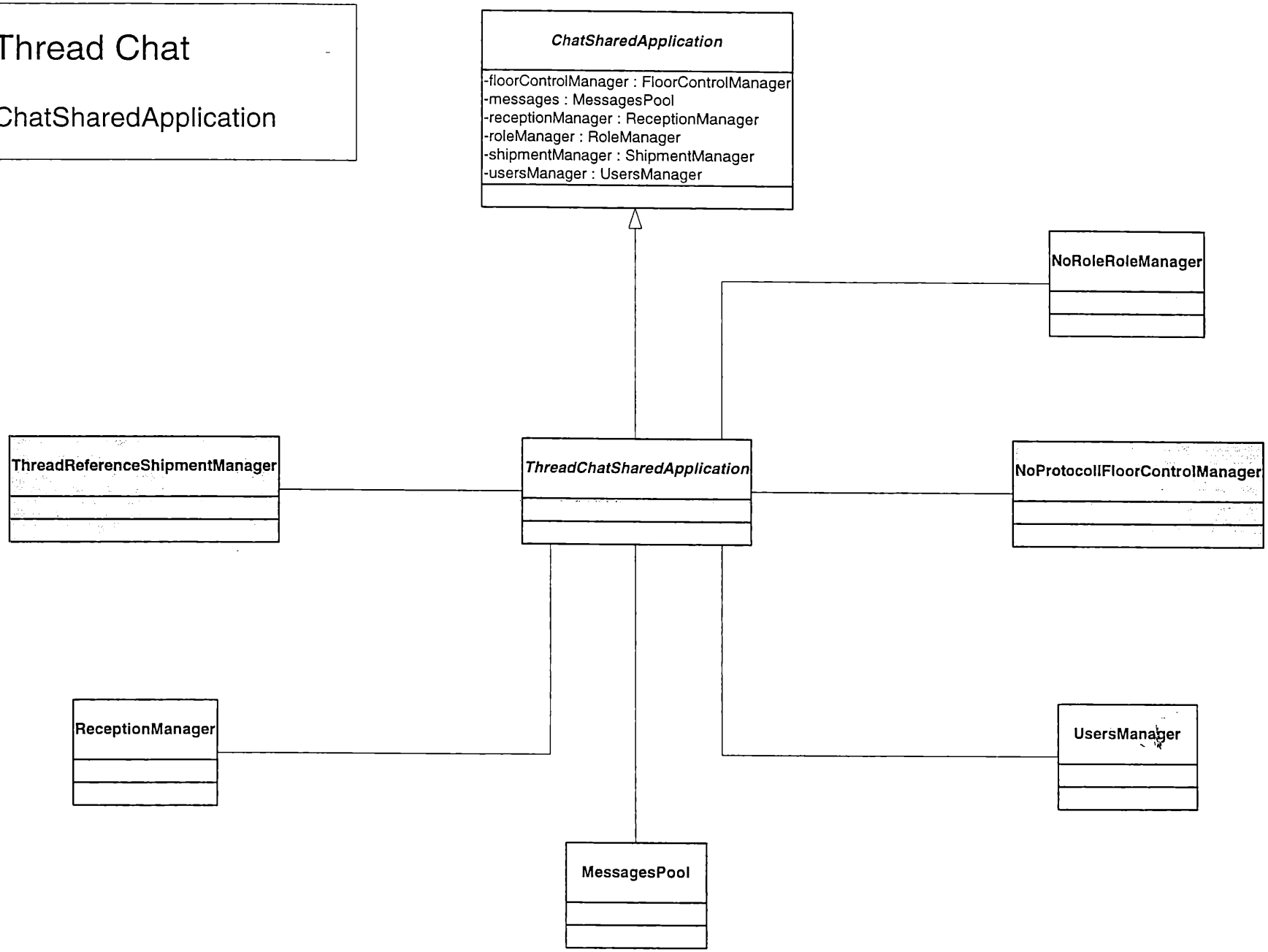
Snap Chat

SimpleChatLocalApplication



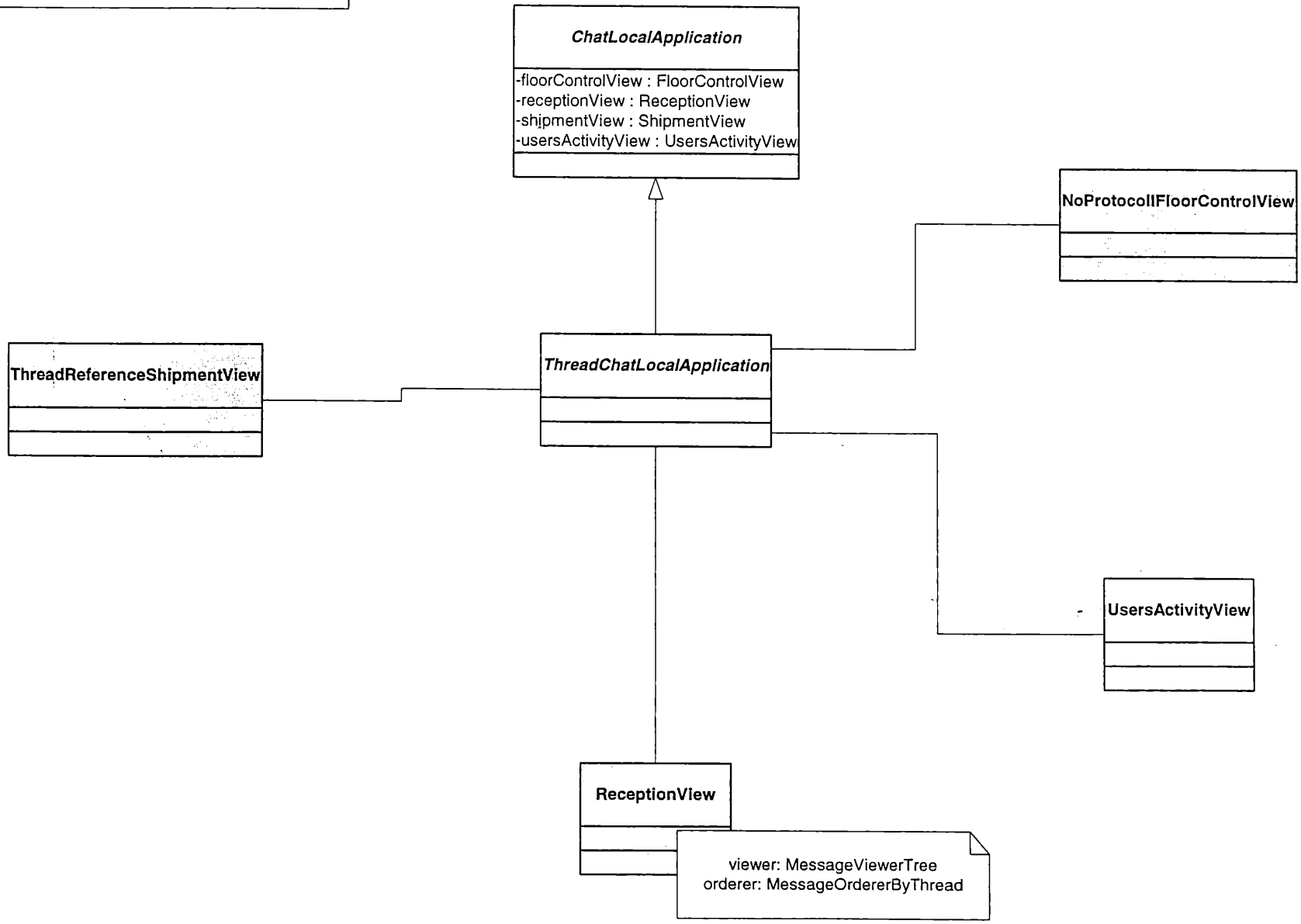
Thread Chat

ThreadChatSharedApplication



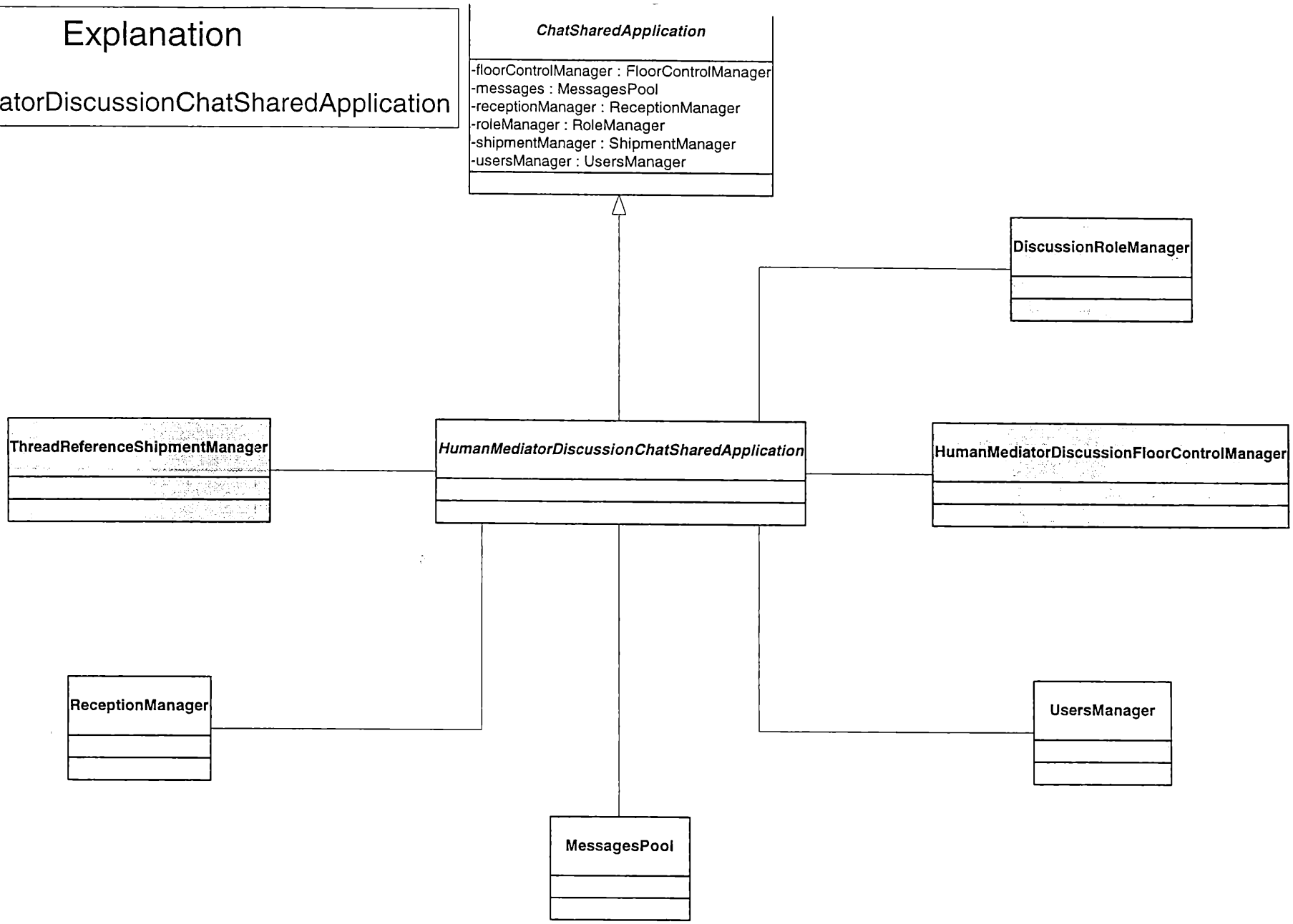
Thread Chat

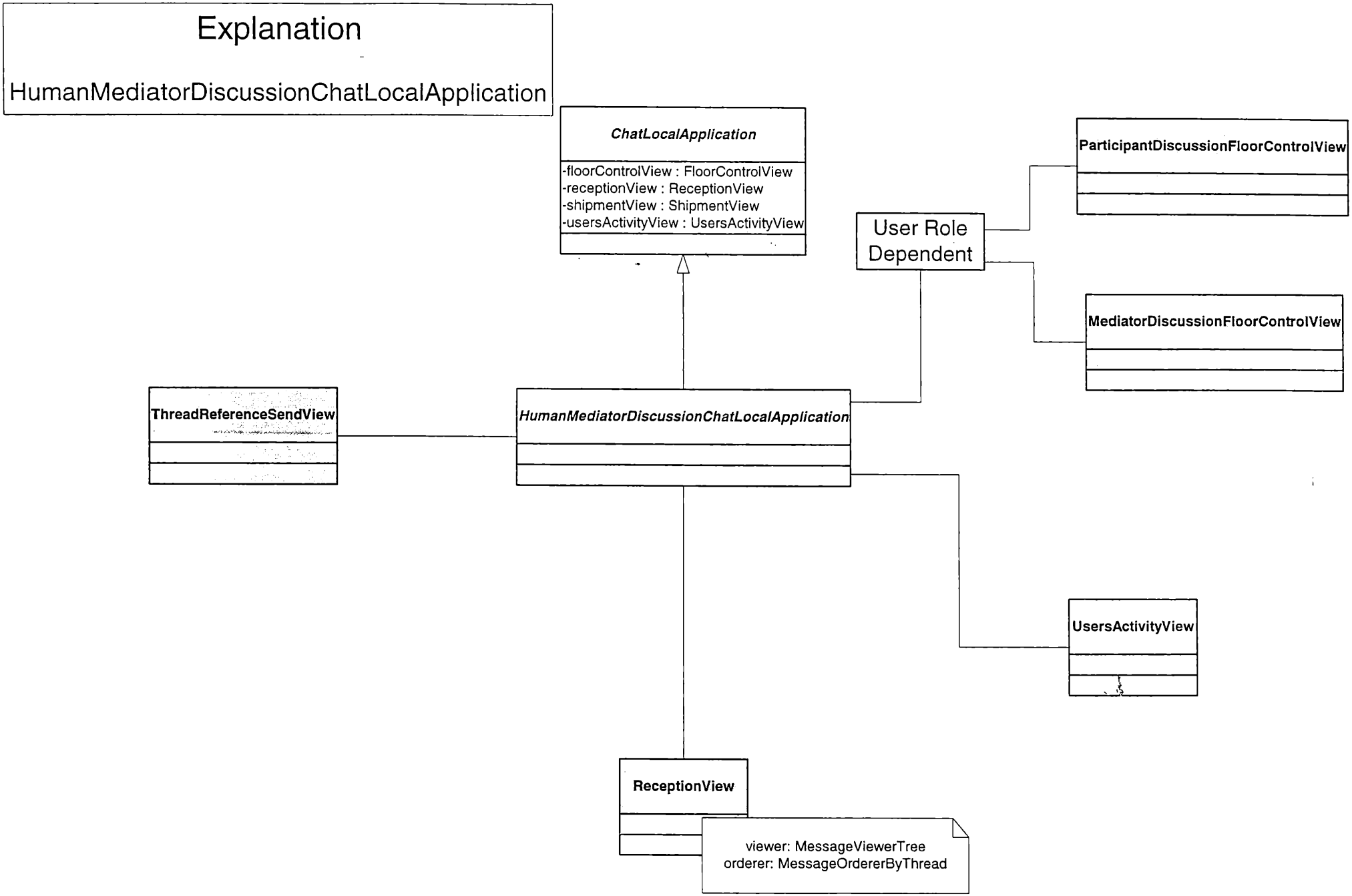
ThreadChatLocalApplication



Explanation

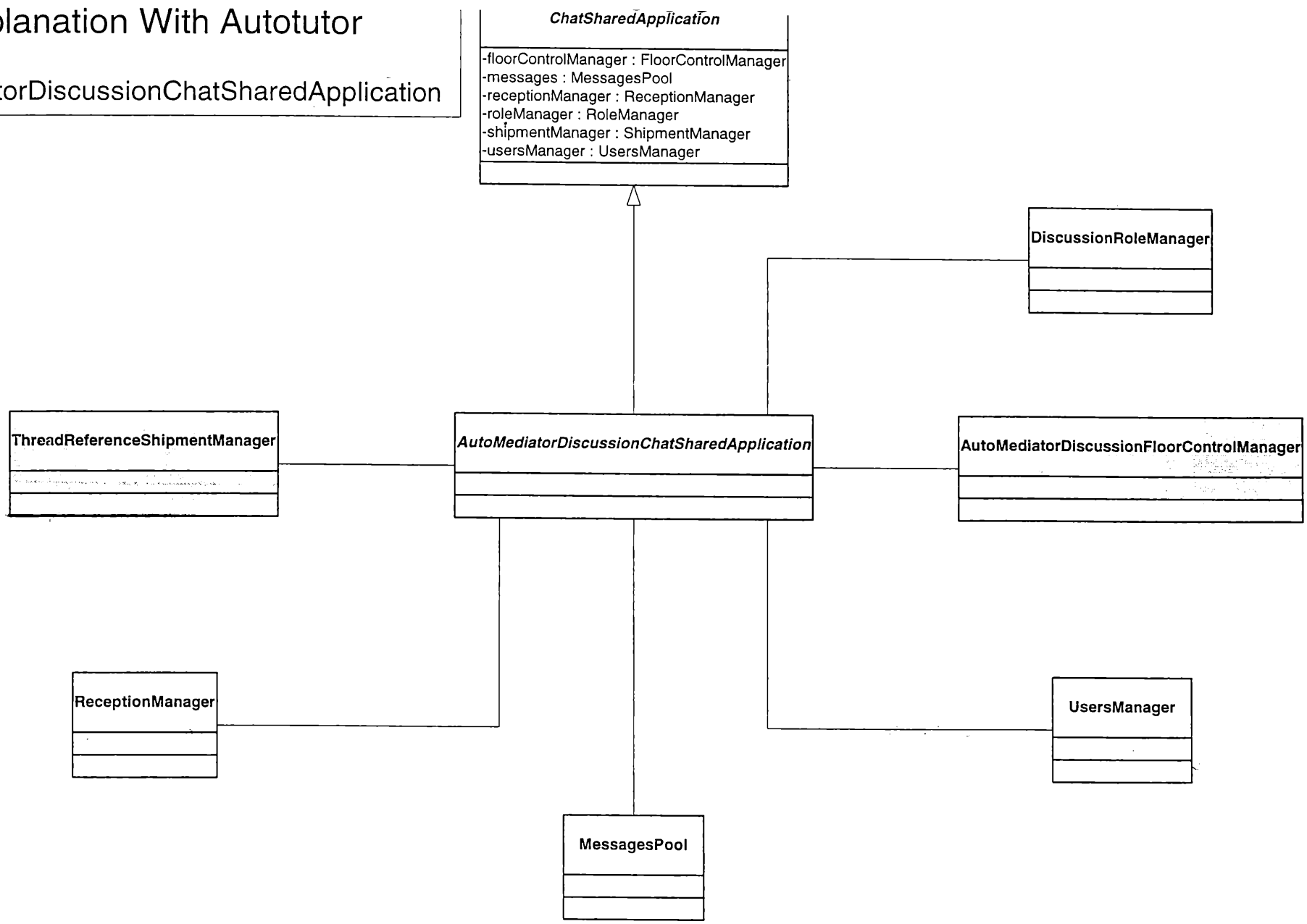
HumanMediatorDiscussionChatSharedApplication




Explanation
HumanMediatorDiscussionChatLocalApplication
ChatLocalApplication
-floorControlView : FloorControlView
-receptionView : ReceptionView
-shipmentView : ShipmentView
-usersActivityView : UsersActivityView
ParticipantDiscussionFloorControlView
User Role Dependent
MediatorDiscussionFloorControlView
ThreadReferenceSendView
HumanMediatorDiscussionChatLocalApplication
UsersActivityView
ReceptionView
viewer: MessageViewerTree
orderer: MessageOrdererByThread

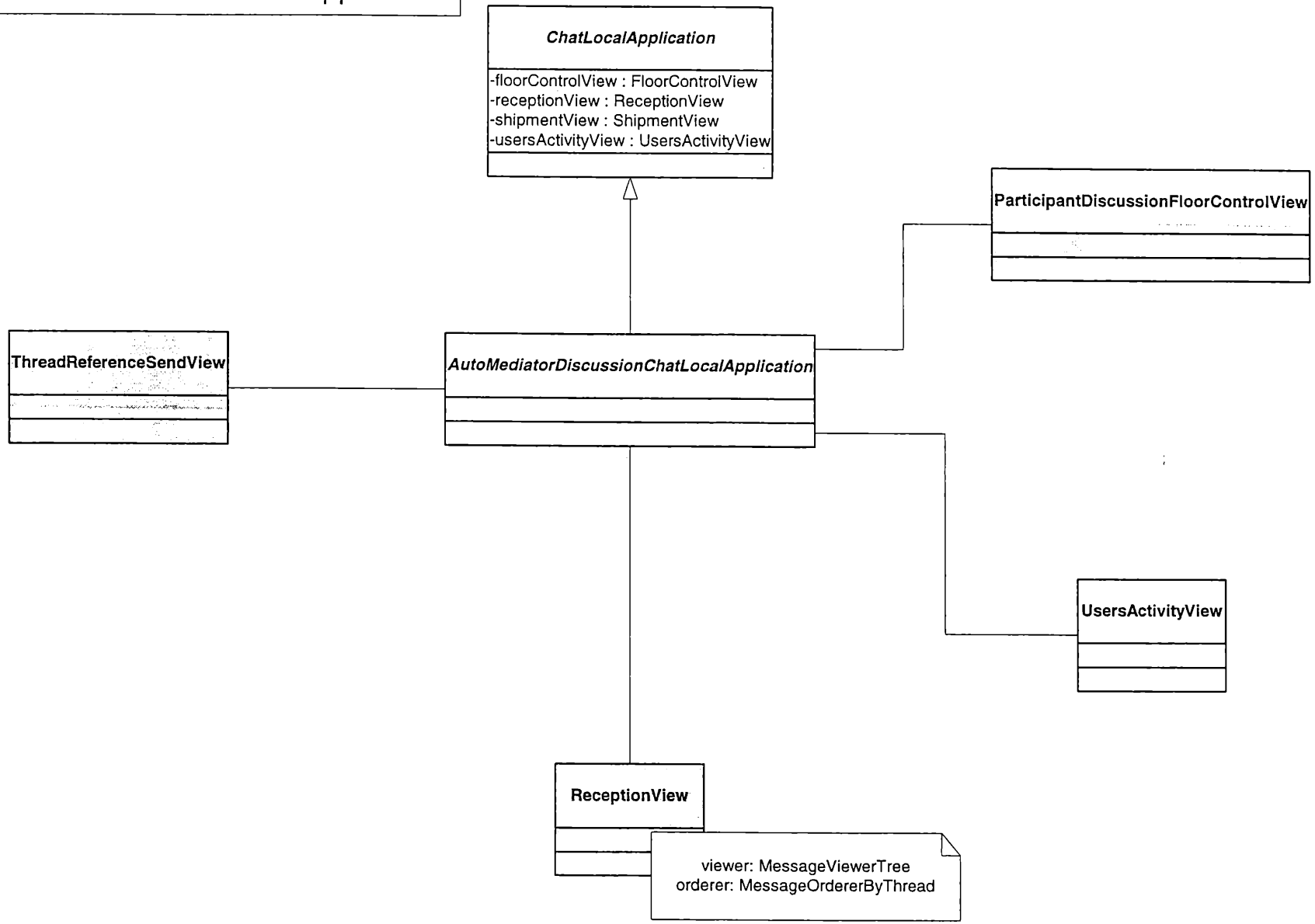
Explanation With Autotutor

AutoMediatorDiscussionChatSharedApplication



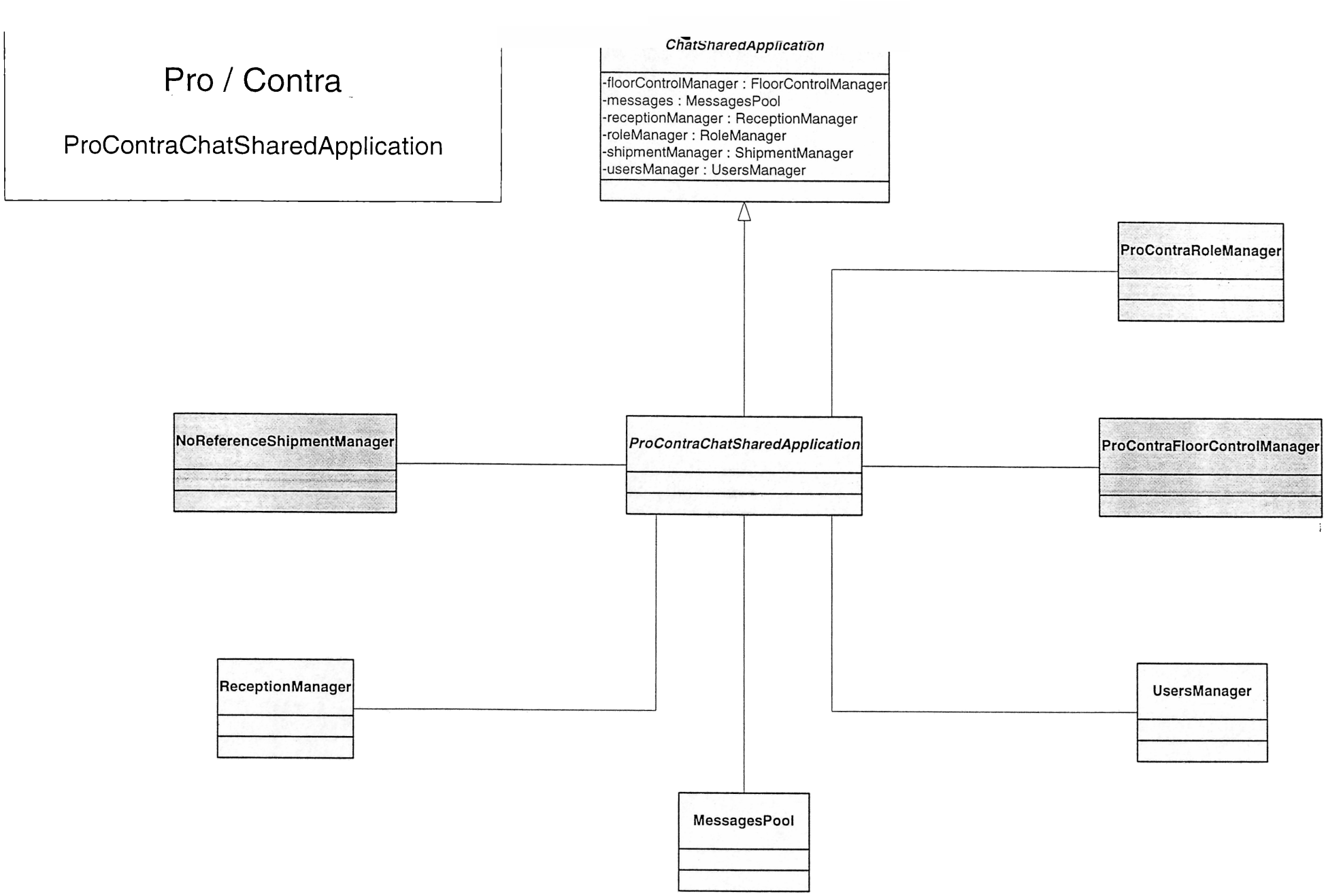
Explanation With Autotutor

AutoMediatorDiscussionChatLocalApplication



Pro / Contra

ProContraChatSharedApplication



Pro / Contra

ProContraChatLocalApplication