Interacción Orientada a Aspectos en Entornos Multiorganizacionales*

Rafael Corchuelo† José A. Pérez‡ Antonio Ruiz–Cortés§

Abstract

Unfortunately, current proposals that focus on separation of concerns assume that objects interact by means of object-oriented method calls, which implies that they embed interactions with others into their functional code. This makes them dependent on this interaction model, and makes it difficult to reuse them in a context in which another interaction model is more suited. In this paper, we show that functionality can be described separately from the interaction model used, which helps enhance reusability of functional code and coordination patterns. In order to show that it is feasible, we adapted the multiparty interaction model to the context of Multi-Organisational Web-Based Systems and developed a class framework to build business objects whose performance rates comparably to a handmade implementation; the development time, however, decreases significantly.

Keywords: *Aspect orientation, interaction models, multiparty interactions.*

Resumen

Por desgracia, las propuestas actuales de separación de aspectos asumen que los objetos interactúan mediante invocación de métodos, lo que implica que tienen que embeber las interacciones dentro del código funcional. Esto los hace dependientes de este modelo de interacción y hace difícil reutilizarlos en un contexto en el que otro modelo de interacción sea más adecuado. En este artículo mostramos que la funcionalidad puede ser descrita de forma independiente al modelo de interacción usado, lo que mejora la facilidad de reutilización del código funcional y los patrones de coordinación. Para demostrar que es viable, hemos adaptado el modelo de interacción entre múltiples participantes al contexto de los sistemas multiorganizacionales basados en la web y hemos desarrollado un *framework* de clases que permite construir objetos de negocio cuyo rendimiento es comparable al de una implementación artesanal; el tiempo de desarrollo, por el contrario, se reduce de forma muy significativa.

Palabras clave: *Orientación a aspectos, modelos de interacción, interacciones multipartitas.*

*Este trabajo está subvencionado por la Comisión Interministerial de Ciencia y Tecnología de España (TIC2000-1106-C02-01)

†Departamento de Lenguajes y Sistemas Informáticos. Escuela Técnica Superior de Ingenieros Informáticos. Universidad de Sevilla, España. e–mail: corchu@lsi.us.es

‡Departamento de Lenguajes y Sistemas Informáticos. Escuela Técnica Superior de Ingenieros Informáticos. Universidad de Sevilla, España. e–mail: jperez@lsi.us.es

§Departamento de Lenguajes y Sistemas Informáticos. Escuela Técnica Superior de Ingenieros Informáticos. Universidad de Sevilla, España. e–mail: aruiz@lsi.us.es

1 Introducción

Los sistemas distribuidos se consideran hoy en día piezas clave en el desarrollo de sistemas multiorganizacionales. Muchas compañías punteras se han dado cuenta de que Internet es mucho más que una zona de compras y ventas y ofrece innumerables posibilidades de triunfar en el mundo de la web. Conforme Internet se asienta, las compañías están encontrando nuevas formas de sacar partido a las capacidades que ofrece y existe una demanda creciente de frameworks [14, 15, 16] para construir sistemas multiorganizacionales basados en la web (MOWS, *Multi-Organisational Web-based systems*) [11, 12] a unos precios razonables. Estos frameworks describen generalmente los MOWS como colecciones de componentes que implementan un conjunto de objetos de negocio que encapsulan un protocolo semántico para realizar una unidad de trabajo lógico: transferir dinero, procesar una factura, actualizar el registro de un cliente, etcétera. Desde un punto de vista abstracto, los objetos de negocio modelan actores y datos reales, por lo que permiten mejorar la comunicación entre desarrolladores y clientes y ayudan a reducir los costes de desarrollo [17].

En teoría, los programadores sólo tienen que preocuparse por la funcionalidad que proporcionan. Por desgracia, esta visión es muy idealista puesto que los objetos de negocio deben preocuparse de aspectos tales como la sincronización, la persistencia o la replicación y en raras ocasiones están aislados dentro de un sistema. Es decir, constituyen la parte funcional de un protocolo semántico y deben trabajar con otros para conseguir un objetivo común. La separación de aspectos ha sido presentada como una herramienta muy prometedora para mejorar la modularidad, facilidad de comprensión y de reutilización del código funcional. De hecho, esta es la piedra angular del nuevo paradigma de programación orientada a aspectos (AOP) [13, 23, 25].

Desde los primeros trabajos en esta materia, la sincronización [25], la distribución [25, 36], la seguridad [37], la coordinación [1, 32], la persistencia [24] o la replicación [5] han sido considerados aspectos claros que se pueden tratar usando lenguajes específicos. Por desgracia, ninguna de las propuestas orientadas a aspectos actuales logran separar la funcionalidad de la interacción con otros objetos en un sistema, es decir, el mecanismo mediante el cual dos o más objetos entran en contacto y pueden desarrollar conjuntamente una parte del trabajo. Tal y como mostramos en la sección 5, las propuestas actuales están basadas en variaciones del modelo de interacción clásico de paso de mensajes y embeben las llamadas a métodos en el código funcional de los objetos, lo que impide a los desarrolladores adaptar su código con facilidad si el modelo de interacción debe ser cambiado. Además, los protocolos semánticos suelen estar codificados dentro del código funcional, lo que hace difícil también su reutilización en escenarios similares [1, 21].

Más allá de la comunicación punto a punto, existen otros muchos modelos de interacción [28], algunos de los cuales también proporcionan mecanismos de sincronización y coordinación. Cada uno tiene sus propias ventajas e inconvenientes, de forma que no existe ningún modelo de interacción universalmente aceptado, sino un amplio abanico de posibilidades. Por lo tanto, parece razonable que existan lenguajes orientados a aspectos que traten el problema de la interacción. Estos lenguajes mejorarían la facilidad de reutilización del código funcional y de los patrones de coordinación. Además, permitirían a los programadores decidir qué modelo de interacción se adapta mejor a sus necesidades, lo que tiene un efecto directo sobre la facilidad de comprensión, de mantenimiento y evolución. Además, esta tecnología para la interacción de objetos debería ser aplicada de forma automática de forma que los objetos de negocio sean tan reutilizables como resulte posible [1, 21].

Esta motivación está avalada por resultados previos de investigación de otros autores. En [27], por ejemplo, se indica que gran parte de la evolución, reutilización e integración del software es de naturaleza inesperada. No obstante, esto no es necesariamente debido a un pobre diseño, sino al hecho de que la tecnología cambia de una forma tan rápida que es imposible predecir los caminos a través de los que evolucionará. Seleccionar un modelo

de interacción adecuado es una decisión importante puesto que la misma funcionalidad encapsulada en una clase podría ser integrada de una forma más simple en un escenario usando paso de mensajes, pero espacios de tuplas en otro. Por lo tanto, tener una forma de cambiar el modelo de interacción mientras se preserva la funcionalidad, parece razonable.

Hasta donde alcanza nuestro conocimiento, la separación entre funcionalidad e interacción es algo nuevo e innovador que no ha sido tratado hasta ahora en el contexto de la orientación a aspectos. En este artículo demostramos que la idea es viable presentando un modelo de interacción para MOWS denominado *Open Multiparty Interaction Model* y un *framework* de clases que permite implementar nuestras ideas de forma muy efectiva. El resto del trabajo se organiza como sigue: en la sección 2 demostramos que nuestra idea puede ir más allá de la invocación de métodos y presentamos el modelo de interacción al que aludimos previamente; en la sección 3 presentamos el *framework* de clases que hemos diseñado para implementar la propuesta; en la sección 4 demostramos que el desarrollo de un MOWS puede beneficiarse del uso de nuestro *framework* y lo evaluamos teniendo en cuenta el rendimiento y el tiempo de desarrollo; en la sección 5 analizamos el trabajo de otros autores y concluimos que ninguno de ellos intenta separar el modelo de interacción de otros aspectos; finalmente, presentamos nuestras conclusiones en la sección 6.

2 Un modelo de interacción de alto nivel para MOWS

Un MOWS se puede ver como un sistema compuesto de varios objetos de negocio de grano grueso que residen en diferentes organizaciones y necesitan cooperar frecuentemente. Las primitivas cliente/servidor como el paso de mensajes o la llamada a procedimientos remotos son el estándar industrial *de facto* para la interacción entre objetos en este contexto. Por desgracia, las soluciones basadas en estas primitivas son difíciles de llevar a cabo en un MOWS en el que varios objetos tienen que cooperar simultáneamente para llegar a cabo una tarea [19]. Ejemplos habituales son la transferencia de dinero de una cuenta a otra mediante un cajero automático (tres objetos), el pago de impuestos a través de la Red (el contribuyente, la Hacienda pública y la Fábrica Nacional de Moneda y Timbre en el caso de España), filtrado en el comercio electrónico (un cliente, un sistema de filtrado y diversos proveedores de servicio), etcétera. La tecnología actual proporciona servidores de transacciones para agrupar interacciones individuales, pero no existe una clara separación entre el problema a solucionar y la tecnología subyacente o el protocolo necesario para coordinar los objetos involucrados en una transacción.

La mayor parte de los métodos de análisis o diseño orientados a objetos reconocen la necesidad de coordinar a varios objetos al mismo tiempo utilizando diferentes términos: diagramas de objetos [4], modelos de procesos [6], conexiones de mensaje [7], grafos de colaboración [38], diagramas de escenario [33] o colaboraciones [35]. Recientemente, esta necesidad también ha sido identificada en el campo de las sociedades multiagentes [3, 26]. El modelo de interacción subyacente es, sin embargo, llamado a métodos en todos los casos, lo que significa que los diseñadores aún tienen que descomponer colaboraciones complejas en secuencias de llamadas.

Por desgracia, los lenguajes de programación actuales no ofrecen un soporte adecuado que permita a varios objetos interactuar simultáneamente, aunque se ha llevado a cabo una gran cantidad de investigación en este campo [28]. JavaSpaces [20] es una de las propuestas más actuales. Proporciona mecanismos mediante los cuales un objeto puede recuperar o introducir otros objetos en un espacio de tuplas compartido, lo que permite desarrollar protocolos semánticos en forma de flujos de objetos que se mueven a través de uno o varios espacios de tuplas. JavaSpaces es perfecto para aquellas aplicaciones en las que los flujos de trabajos son adecuados, pero no es suficiente en general, principalmente en el campo de los MOWS o las aplicaciones de comercio electrónico en particular. La razón es que muchos

problemas en este contexto precisan de varios objetos que cooperen simultáneamente y transformar estas colaboraciones en flujos de trabajo no es nada sencillo.

Por ejemplo, supongamos que deseamos diseñar un sistema de tarjetas de débito [34]. Este sistema se compone de varios puntos de venta, varios ordenadores que tienen las cuentas de los vendedores y las de los clientes; el objetivo es diseñar un objeto de negocio que sea responsable de transferir dinero de una cuenta de un cliente a una cuenta de un vendedor a través de un punto de venta cada vez que el primero usa su tarjeta de débito. Con JavaSpaces, por ejemplo, es relativamente sencillo desarrollar una solución para resolver este problema, pero tendría dos inconvenientes fundamentales: primero, la transacción completa se puede ver como una interacción que coordina a tres objetos, pero este modelo de interacción lleva a una solución en la que es necesario descomponer este evento en varias llamadas a métodos que deben ser coordinadas mediante un protocolo específico; además, este protocolo puede ser reutilizable en situaciones similares, pero la separación con JavaSpaces no es suficiente, por lo que no se pueden reutilizar con facilidad [23].

2.1 Nuestra propuesta

Sobre la base de la discusión previa, queda clara la necesidad de contar con un modelo de interacción capaz de coordinar la acción conjunta de varios objetos. Para diseñar este modelo partimos del Modelo de Interacción entre Múltiples Participantes [18], que es muy adecuado para capturar la esencia de estos problemas. Proporciona un alto grado de abstracción que permite ignorar por completo cuál es la forma en que varios objetos se sincronizan simultáneamente o intercambian información. Esto, además, abre las puertas a una mayor facilidad de adaptación de las aplicaciones puesto que los algoritmos utilizados quedan a un nivel de abstracción menor y pueden ser cambiados con facilidad sin afectar al sistema diseñado.

Este modelo ha atraído la atención de un buen número de investigadores que se han centrado en aspectos de implementación [9, 19, 39]. No obstante tiene varios defectos que dificultan su aplicación a los MOWS: el modelo no tiene en cuenta la existencia de objetos pasivos, los objetos deben ser conocidos en tiempo de compilación y tienen que tener acceso al estado local de otros objetos para poder comunicarse con ellos. Esta es la razón por la que hemos desarrollado una versión avanzada del mismo a la que nos referimos como *Open Multiparty Interaction Model* (OMIM).

Para soportar OMIM, hemos diseñado un lenguaje llamado CAL [8] y un conjunto de algoritmos que permiten implementarlo [29, 30, 31]. En las secciones siguientes resumimos el trabajo que llevamos hecho; en la sección 3, presentamos el *framework* que hemos desarrollado para implementarlo.

2.2 Soporte lingüístico

En esta sección describimos CAL someramente mediante el ejemplo del sistema de tarjetas de débito presentado previamente. Este problema se puede describir fácilmente utilizando nuestro modelo puesto que realmente se puede reducir a una interacción tripartita entre un punto de venta, una cuenta de cliente y una cuenta de vendedor. La figura 1 muestra una descripción del sistema en CAL que analizamos en las secciones siguientes.

2.2.1 Descripción de interacciones

Las interacciones se definen usando la siguiente sintaxis:

interaction ⟨*nombre*⟩[⟨*objetos participantes*⟩](⟨*ranuras*⟩)
    where ⟨*permisos de lectura/escritura*⟩

Cada una tiene un nombre diferente, algunos participantes, algunas ranuras y un conjunto de permisos de lectura/escritura.

```
interaction transfer[PointOfSales pos; Account source, dest]
      (float sum; boolean approval) where
      pos writes sum, reads approval;
      source writes approval, reads sum;
      dest reads approval, sum;


behaviour PointOfSales requires
   interface IPointOfSales;
{
*[
  Wait_For_Sale();
  transfer[pos!self, source!Get_Customer_Account(), dest!Get_Merchant_Account()]
  {
  sum = Get_Price();
  Report_Result(approval);
  }
]
}

behaviour Account requires
interface IAccount;
{
*[
  transfer[source!self]
  {
  approval = Authorise_Payment(sum);
  [approval → Charge(sum);];
  }
  []
  transfer[dest!self]
  {
   [approval → Pay_In(sum);];
  }
  ]
}
```

Figure 1: Una descripción en CAL del sistema de tarjetas de débito.

En el ejemplo en la figura 1 se ha definido una interacción tripartita denominada transfer que permite interactuar a un objeto que interpreta el papel pos y dos cuentas que interpretan los papeles source y dest. Esta interacción se convierte así en el canal mediante el cual pueden coordinar sus actividades para conseguir transferir fondos desde la cuenta origen (source) hasta la cuenta destino (dest) mediante el punto de ventas (pos).

Las interacciones tienen un estado local que se compone de varias ranuras. transfer, por ejemplo, tiene dos ranuras denominadas sum y approval. La primera se utiliza para almacenar la cantidad de dinero a transferir y la segunda es un indicador de si la cuenta origen tiene saldo suficiente como para transferir esa cantidad a la cuenta destino. Estas ranuras dan

```
interface IPointOfSales {
   void Wait_For_Sale();
   float Get_Price();
   oid Get_Customer_Account();
   OID Get_Merchant_Account();
   void Report_Result(boolean done);
}
```

```
interface IAccount {
   void Charge(float sum);
   void Pay_In(float sum);
   boolean Authorise_Payment(float sum);
}
```

Figure 2: Interfaces requeridas por los comportamientos en la figura 1.

lugar a un estado local que simula el estado global temporal del modelo de interacción entre múltiples participantes básico [19]. La diferencia más importante es que un objeto ya no necesita tener acceso al estado local de otros para obtener la información que necesita.

Los permisos de lectura/escritura indican qué participante puede leer o escribir cada ranura. En nuestro ejemplo, el punto de venta es responsable de almacenar la suma a transferir en la ranura sum, mientras que tan sólo necesita leer la ranura approval para mostrar el mensaje apropiado en la pantalla; la cuenta que interpreta el papel source puede leer la ranura sum para decidir si puede transferir la suma indicada y escribe en la ranura approval para almacenar su decisión; finalmente, la cuenta destino puede leer ambas ranuras, pero no tiene permiso para escribir en ninguna de ellas.

Un objeto puede ofrecer participación en una o más interacciones al mismo tiempo. En cada oferta debe indicar qué papel quiere interpretar y puede establecer restricciones sobre qué otros objetos deben interpretar los restantes papeles. Una interacción se puede ejecutar siempre y cuando exista un conjunto de objetos que esté de acuerdo en interaccionar. Generalmente nos referimos a esa interacción y el conjunto de objetos que la pueden ejecutar como una habilitación.

Dado que es necesario garantizar la exclusión mutua, un objeto no puede ejecutar más de una interacción al mismo tiempo. No obstante, como un objeto puede ofrecer participación en diversas interacciones al mismo tiempo, puede encontrarse en más de una habilitación. Por esta razón, cuando dos o más habilitaciones comparten objetos, no pueden ejecutarse al mismo tiempo. El conjunto de habilitaciones que no se pueden ejecutar se dice que son rechazadas.

2.2.2 Descripción de los comportamientos

Como hemos visto, los objetos participantes en una interacción deben interpretar papeles concretos en ella. La sintaxis que utilizamos para describirlos es la siguiente:

behaviour ⟨*nombre*⟩
  requires ⟨*interfaces*⟩
{
⟨*descripción de comportamiento*⟩
}

Cada descripción requiere que los objetos sobre los que se aplica implementen un conjunto de interfaces. En el ejemplo de la figura 1 se han mostrado dos comportamientos: PointOfSales, que describe el la forma en que se comportan los puntos de venta, y Account, que describe el comportamiento de las cuentas bancarias, ya sea como deudoras o como acreedoras. PointOfSales requiere la interfaz IPointOfSales descrita en la figura 2. Las operaciones de esta interfaz permiten a un objeto esperar a que se produzca la siguiente venta (Wait_For_Sale) y obtener su importe (Get_Price), así como obtener referencias a las cuentas

del cliente y el vendedor (Get_Customer_Account, Get_Merchant_Account) y mostrar un mensaje en pantalla con los resultados de la operación (Report_Result). De manera similar, Account requiere la implementación de una operación para retirar dinero de una cuenta (Charge), para ingresar dinero en ella (Pay_In) o para autorizar un pago (Authorise_Payment).

Las operaciones requeridas por un comportamiento son aquéllas cuya ejecución se coordina mediante interacciones multipartitas. Para modelar la forma en que un punto de venta y dos cuentas cooperan utilizamos instrucciones de la forma I[participant_list]{comm_stat}, en donde I es el nombre de una interacción, la lista entre corchetes identifica los objetos que se desea hacer participar en la interacción y comm_stat es una lista de instrucciones de comunicación que involucra las ranuras de la interacción I. Los participantes en la interacción se indican mediante expresiones de la forma role!id, lo que significa que el papel role debe ser interpretado por el objeto identificado por id. La expresión role!self se utiliza para indicar que el papel role será interpretado por el objeto que ejecuta la instrucción de interacción. Si no se indica nada para un papel, se asume que cualquier objeto puede interpretarlo.

Por lo tanto, el comportamiento de un punto de venta se puede resumir así: es un bucle infinito en el que primero espera a que comience una nueva operación de venta y entonces intenta ejecutar la interacción transfer junto a los objetos que modelan la cuenta del cliente y del vendedor. Si se llega a ejecutar esta interacción, el punto de venta ejecuta entonces su código de comunicación, que consiste en almacenar la suma a transferir en la ranura sum y mostrar un mensaje en su pantalla. El comportamiento Account también consiste en un bucle infinito en el que la participación en la interacción transfer se ofrece tanto como cuenta de origen como de destino de la transferencia. Si se ejecuta la interacción en la que interpreta el primer papel, entonces comprueba si puede afrontar el cargo, almacena el resultado en la ranura approval y actualiza su saldo convenientemente. Si se ejecuta la otra interacción, entonces simplemente lee la ranura approval y actualiza el saldo.

Evidentemente, una interacción multipartita retrasa la ejecución de un objeto que intenta leer una ranura hasta que ha recibido un valor por parte de otro participante. Es decir, un punto de venta que intenta ejecutar la instrucción Report_Result(approval), es retrasado en su ejecución hasta que la cuenta origen de la transferencia escribe su decisión en la ranura approval. Por lo tanto, las instrucciones de comunicación se ejecutan en una región crítica en la que no pueden producirse condiciones de carrera.

2.2.3 Aplicación de comportamientos a clases

Los patrones de comportamiento son abstractos porque describen cómo un objeto que implementa un conjunto de operaciones puede cooperar con otros. Estas operaciones son también abstractas y, a menudo, necesitan ser adaptadas antes de aplicar un comportamiento sobre una clase concreta.

CAL proporciona un mecanismo simple para adaptar operaciones que se muestra en la figura 3. En este ejemplo, el comportamiento Account ha sido aplicado sobre una clase Java llamada bankAccount. Si se fija, esta clase no proporciona las operaciones que requiere el comportamiento, aunque no es difícil adaptarlas. Por ejemplo, la operación Authorise_Payment(sum) se puede adaptar fácilmente mediante la expresión getBalance() $\geq$ sum.

Esta adaptación nos permite escribir comportamientos completamente abstractos y reutilizables. Esto es importante puesto que en otros lenguajes orientados a aspectos como COOL [25], RIDL [25] o AspectJ [22], los aspectos hacen referencia a las clases sobre las que se aplican por nombre, lo que provoca dependencias entre ellos y hace difícil reutilizar los primeros.

```
class bankAccount {
   private float balance;

   public void updateBalance(float sum) { balance += sum; }
   public boolean getBalance() { return balance; }
}

map behaviour Account onto class bankAccount where
      Charge(sum) = updateBalance(-sum);
      Pay_In(sum) = updateBalance(+sum);
      Authorise_Payment(sum) = (getBalance() ≥ sum);
```

Figure 3: Aplicación del comportamiento Account sobre la clase bankAccount.

3 Nuestro *framework*

Hemos diseñado un *framework* de clases que ofrece todos los servicios necesarios para implementar CAL[1] Nuestros objetivos principales a la hora de diseñarlo fueron los siguientes:

1. Debería ser extensible de forma que se pudieran incorporar en él nuevos middlewares y algoritmos de coordinación de una forma simple. Esto mejora las posibilidades de elección del diseñador a la hora de producir la versión final de sus objetos de negocio.

2. Debe tener en cuenta la existencia de objetos pasivos y tratarlos adecuadamente. El modelo de interacciones entre múltiples participantes tradicional asume que todos los objetos son activos y cuentan con su propio hilo de ejecución, lo que no suele ser cierto en el contexto de los MOWS.

3. Debe permitir integrar otros aspectos ortogonales con facilidad.

La figura 4 muestra una fotografía de un sistema en ejecución que ilustra la arquitectura de nuestra propuesta. Se compone de los elementos siguientes:

El *gatekeeper*: Es uno de los elementos más importantes puesto que es responsable de tareas como las políticas de seguridad, facturación, generar y gestionar los UUID, localizar los coordinadores de interacción, etcétera.

Coordinadores de interacción: Son responsables de detectar qué interacciones están habilitadas y de arbitrar entre aquéllas que son conflictivas, es decir, comparten objetos.

Representantes: Los objetos del usuario se consideran entidades externas al *framework* que utilizan representantes para interactuar. Esto permite establecer una clara separación entre la funcionalidad y los detalles de coordinación, aunque también simplifica el diseño del *framework* ya que éste sólo tiene que tratar con los representantes. Esto significa que los objetos de usuario puede tener funcionalidad pura o han podido ser previamente mezclados con otros aspectos. Por ejemplo, un objeto que tenga restricciones de sincronización y de persistencia puede ser implementado utilizando COOL [25] y la propuesta de [5], pero estos detalles son irrelevantes para el *framework*.

Gestores de comunicación: Son los responsables de gestionar la comunicación entre los objetos que participan en una misma interacción, es decir, permiten la comunicación,

[1] Este *framework* está disponible solicitándolo a los autores. Envíe una carta electrónica a jperez@lsi.us.es si desea obtener una copia.

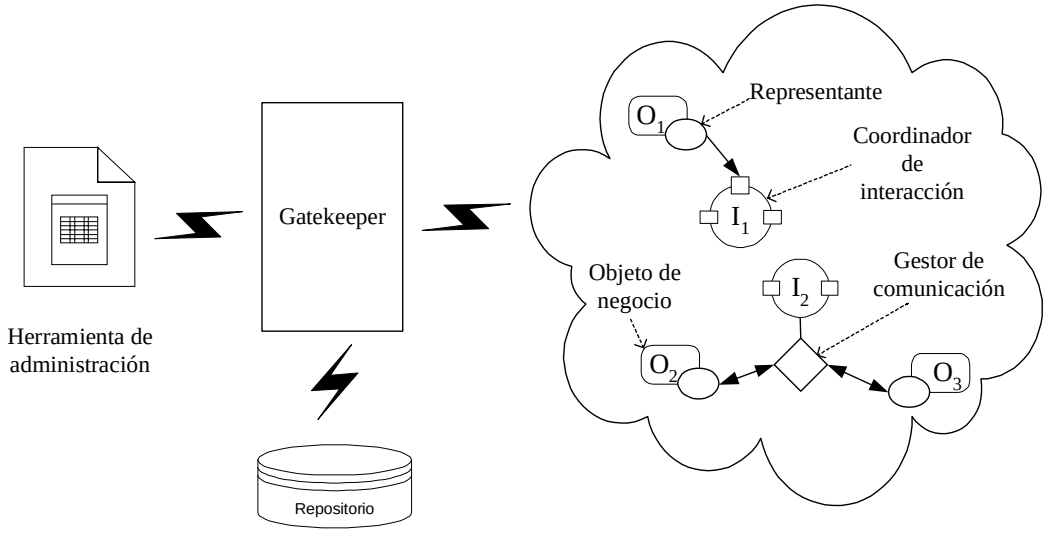

Figure 4: La arquitectura de nuestra solución.

tienen en cuenta los fallos que se puedan producir, etcétera. Dado que es posible que existan varias ocurrencias de la misma interacción ejecutándose simultáneamente, cada una de ellas tiene su propio gestor de comunicaciones.

A primera vista, podría parecer que el *gatekeeper* es un cuello de botella, pero esto no es así. La razón es que la funcionalidad que ofrece se utiliza sólo cuando aparecen nuevos objetos o interacciones en el sistema o cuando un objeto necesita localizar las referencias a los coordinadores responsables de las interacciones en las que participan. También debemos indicar que nada nos impide crear varios *gatekeepers* que se ejecutan concurrentemente, reduciendo de esta forma el impacto de una caída o fallo del sistema. Sin embargo, dado que todas las copias del componente en ejecución son funcionalmente equivalentes entre sí, solemos referirnos al mismo como "el *gatekeeper*".

También es interesante mencionar que tener representantes para los objetos de usuario no significa ineficiencia puesto que residen en el mismo espacio de memoria que los objetos que representan. Además, separar los aspectos de coordinación de los objetos de negocio en tiempo de ejecución también dibuja una clara línea de separación entre la funcionalidad que encapsulan y la forma en que interactúan con otros. Esto facilita la construcción de *weavers* puesto que los representantes se pueden implementar fácilmente usando una colección de patrones de diseño a la que de forma colectiva se le suele hacer referencia como *Role Object Pattern* [15]. Esto ayuda a mantener separados los distintos contextos en los que un objeto puede participar y facilita el mantenimiento de los mismos.

La figura 5 muestra el aspecto del *framework* que hemos diseñado. Fíjese en que nuestro diseño es lo suficientemente general como para acomodar con facilidad diversos middlewares o algoritmos de coordinación o de comunicación propuestos en la bibliografía, entre ellos los nuestros [30, 29]. El diseñador puede de esta forma decidir cuál es la mejor forma de implementar sus objetos de negocio: si todos los objetos conocen aquéllos con los que van a interactuar se puede utilizar un algoritmo clásico como el de Bagrodia *et al.* [2] o los de Corchuelo *et al.* [9, 10]; en caso contrario será preciso utilizar los algoritmos de Pérez *et al.* [30, 29].

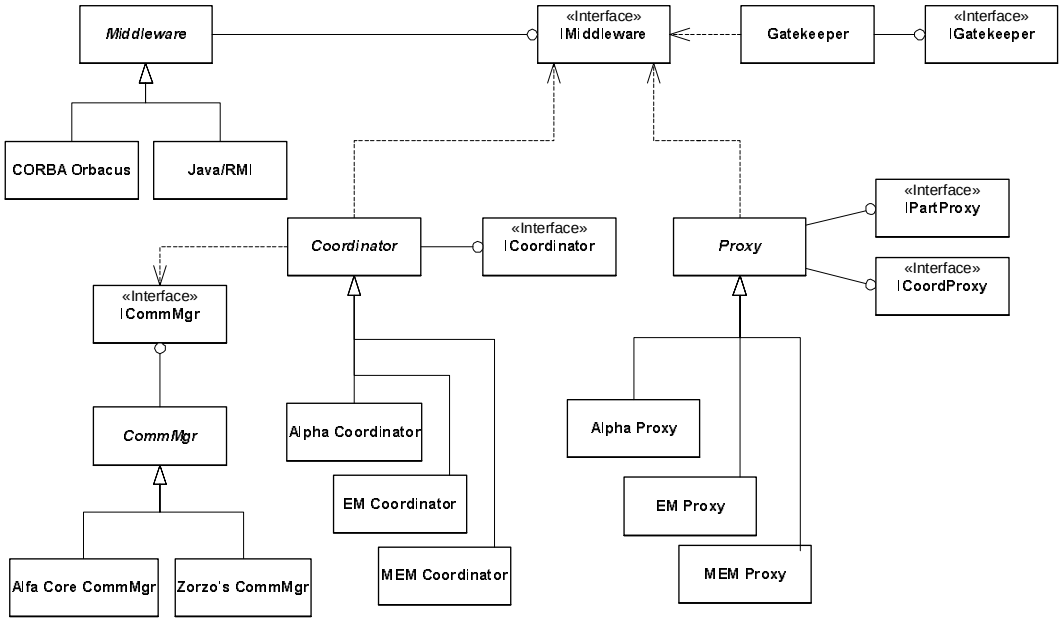

Figure 5: Nuestro *framework* de clases.

4 Evaluación de la propuesta

En las secciones siguientes demostraremos la flexibilidad de nuestra propuesta y su eficiencia viendo de qué forma permite cambiar el modelo de interacción, cómo se pueden integrar aspectos ortogonales y mostrando los resultados de unos experimentos sobre eficiencia y tiempo de desarrollo que hemos llevado a cabo.

4.1 Cambio del modelo de interacción

Nuestra propuesta permite reutilizar fácilmente la funcionalidad de una clase en contextos en los que se utilizan diversos modelos de interacción. Lo ilustraremos mediante un ejemplo en el que reutilizamos la clase bankAccount en un entorno en el que se utiliza JavaSpaces [20] para desarrollar un objeto de negocio que permite cargar las cuentas que sea necesario con una determinada comisión.

La figura 6 presenta el código Java necesario. commissionCharger es un objeto de negocio que vigila continuamente el espacio de tuplas commissionSpace. Cada vez que aparece en él una tupla de la forma comission(oid, p), la retira del mismo, localiza el objeto de la clase bankAccount identificado por oid y le carga el porcentaje de comisión p.

Este ejemplo sencillo demuestra que es posible reutilizar la funcionalidad que ofrece la clase bankAccount de una forma efectiva siempre y cuando se mantenga independiente del modelo de interacción utilizado.

4.2 Integración con aspectos ortogonales

El lenguaje que hemos presentado permite integrar fácilmente otros aspectos de la forma en que se muestra en la figura 7. La idea es mezclar aspectos tales como sincronización intra-objetos o replicación utilizando lenguajes específicos para describirlos de forma que la clase sobre la que apliquemos un comportamiento descrito con CAL ya haya sido mezclada con todos ellos.

```
public class commission implements net.jini.core.entry.Entry {
    public String OID;
    public Float percentage;

    public commission() {}
}
public class commissionCharger {
    protected bankAccount locateAccount (String OID)
    { ...Localizar la cuenta mediante su OID... }

    public void run()
    {
    JavaSpace space = spaceAccessor.getSpace("commissionSpace");
    commission orderTemplate = new commission();
    commission newOrder = null;
    bankAccount account = null;

    try
    {
    while (true)
     {
     newOrder = (commission)space.take(orderTemplate, null, Long.MaxValue);
     account = locateAccount(newOrder.OID);
     if (account.getBalance() > 0)
     account.updateBalance(-account.getBalance() * newOrder.percentage.getFloatValue()));
     }
    }
    except (Exception e)
    {
     e.printStackTrace();
    }
    }
}
```

Figure 6: Un objeto de negocio que carga comisiones usando JavaSpaces.

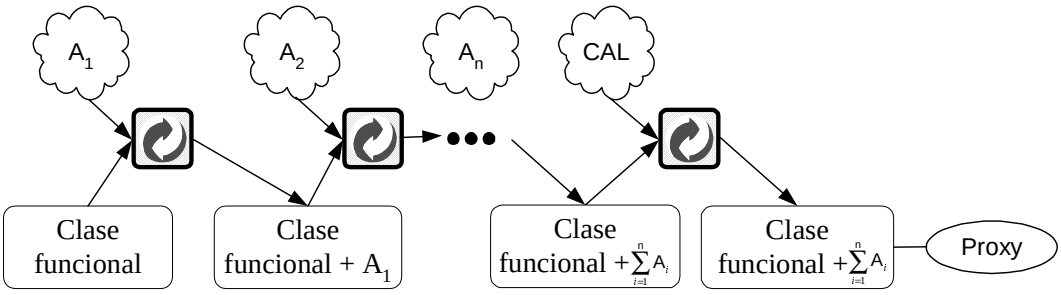

Figure 7: Mezclando aspectos ortogonales con CAL.

Table 1: Patrones de coordinación utilizados para evaluar nuestra propuesta (c.f. [19]).

Clave	Descripción
DCS	El sistema de tarjetas de débito presentado anteriormente.
LE(n)	El problema de la elección del líder, que es un patrón muy habitual en subastas a través de la web. El número de subastantes se da entre paréntesis.
DF(n)	El problema de los filósofos comensales, que es el paradigma de todas aquellas situaciones en las que un objeto necesita tener acceso exclusivo a otros de forma simultánea. El número de filósofos se da entre paréntesis.
MM	El problema de multiplicación de matrices, que es un ejemplo típico de tarea que debe ser dividida en varias subtareas y que requiere de una cuidadosa coordinación para obtener los resultados totales a partir de los resultados parciales.
TH	Las Torres de Hanoy. Aunque este patrón no es frecuente en la práctica, es un buen ejemplo de problema en el que un número relativamente pequeño de objetos tienen que interactuar muy frecuentemente utilizando un patrón de coordinación muy complejo. Sirve para probar nuestro *framework* en una situación extrema.

Por ejemplo, la figura 8 muestra una especificación en COOL que indica que no es posible que dos hilos de ejecución concurrentes ejecuten en método updateBalance simultáneamente o al mismo tiempo que el método getBalance. Podemos usar el *weaver* de COOL para mezclar esta especificación con la clase bankAccount de manera que la clase resultante tenga sincronización intra-objeto. Se forma similar podemos usar los *advices* en AspectJ descritos en [5] para añadirle capacidades de replicación de una forma muy simple.

4.3 Resultados empíricos

Al objeto de evaluar empíricamente nuestra propuesta hemos desarrollado varios experimentos en los que hemos implementado los patrones de coordinación que mostramos en la tabla 1 utilizando nuestra propuesta, AspectJ y Java estándar. Los experimentos fueron desarrollados por equipos de tres estudiantes de cuarto grado. Todos tenían buenos conocimientos en Java, de forma que tan sólo hubo que enseñarles a utilizar nuestro *framework* y AspectJ.

```
coordinator bankAccount {
    selfex updateBalance;
    mutex {updateBalance, getBalance};
}
```

Figure 8: Un aspecto ortogonal mezclado con CAL.

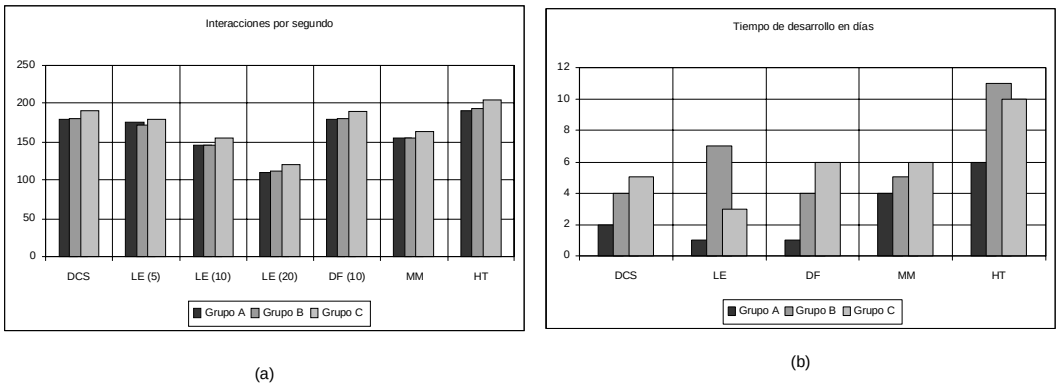

Figure 9: (a) Rendimiento de nuestro *framework*. (b) Tiempo de desarrollo.

Después de formar a nuestros estudiantes, los dividimos en doce grupos de tres personas cada uno. Todos implementaron los patrones indicados, pero los cuarto primeros utilizaron nuestro *framework* (Grupo A), los cuatro siguientes usaron AspectJ (Grupo B) y los restantes usaron Java estándar (Grupo C).

La figura 9.a proporciona resultados empíricos sobre la eficiencia de la implementación que cada grupo desarrolló. Todos los experimentos se ejecutaron en una red de ordenadores equipados con procesadores Pentium a 200 Mhz, 64 MB de RAM y Windows NT 4.0 unidos entre sí mediante una red Ethernet de 10 Mbps. Medimos el número medio de interacciones por segundo que consiguió cada implementación durante una ejecución lo suficientemente larga como para ejecutar 10.000 interacciones.

Como muestra la figura, nuestro *framework* consigue unas 163.12 interacciones por segundo utilizando nuestros propios algoritmos, lo que resulta bastante similar a las implementaciones realizadas con AspectJ o Java. La penalización que introduce nuestro *framework* es de aproximadamente 0.62% con respecto a AspectJ y de 5.34% con respecto a Java. Parece bastante razonable y se ve compensada con la reducción significativa que se produce en el tiempo de desarrollo. Como se muestra en la figura 9.b, este tiempo es de aproximadamente 121.43% menos que usando AspectJ y de aproximadamente 114.29% menos que usando Java.

A partir de estos resultados, podemos concluir que utilizar AspectJ para resolver problemas específicos en el campo de la coordinación no implica una reducción significativa del tiempo necesario para desarrollar un proyecto. Creemos que la razón es que AspectJ no proporciona construcciones específicas para tratar este aspecto, por lo que es necesario emplear la mayor parte del tiempo desarrollando los algoritmos de coordinación, lo que no supone una ventaja respecto a usar Java estándar.

5 Trabajo relacionado

Una de las primeras propuestas que consideró la coordinación como un aspecto separado fue COOL [25]. Este lenguaje proporciona un constructor especial denominado *coordinator* que se puede asociar con una clase concreta y permite describir la política de sincronización de sus métodos. El coordinador de una clase describe todos los posibles estados por lo que pueden pasar sus objetos desde el punto de vista de la coordinación, cómo cambian de estado cuando se invoca un método y también establece restricciones sobre cómo debe ejecutarse una llamada a método dependiendo del estado en que se encuentra el objeto.

COOL se diferencia de nuestra propuesta en que tan sólo es capaz de tratar con la coordinación intra-objeto, es decir, los protocolos que controlan cuándo una llamada a método puede ser ejecutada inmediatamente o debe ser retrasada hasta que se cumpla una determinada condición de sincronización. Por lo tanto, la coordinación entre varios objetos debe ser codificada dentro del código funcional. Además, COOL asume que el modelo de interacción subyacente es llamada a métodos, lo que hace imposible cambiarlo.

Anteriormente, los autores de [21] presentaron una propuesta a la que denominamos LFMOC. Utiliza restricciones multiobjeto implementadas por sincronizadores que permiten describir patrones de coordinación. Conceptualmente, un sincronizador es un objeto que restringe de alguna forma las invocaciones de métodos aceptadas por un conjunto de objetos. Permiten establecer un orden temporal o la atomicidad de un conjunto de llamadas, por lo que esta propuesta mejora COOL substancialmente. ACT [1] es otra propuesta similar a LFMOC. En este caso, se utilizan filtros de composición para diseñar metaobjetos que controlan de qué forma se sincroniza la ejecución de un conjunto de métodos en uno o varios objetos. LFMOC y ACT son similares en esencia, pero se diferencian en las posibilidades expresivas de los lenguajes que utilizan para describir la coordinación, que son mucho mayores en el segundo caso. Sin embargo, LFMOC permite generar los protocolos de bajo nivel necesarios para conseguir la sincronización entre objetos de una forma automática. Por el contrario, estos protocolos deben ser descritos y diseñados por el programador en ACT, aunque pueden ser reutilizados en varias ocasiones.

Nuestra propuesta es similar tanto a LFMOC como a ACT puesto que también permite coordinar grupos de objetos sin necesidad de embeber los protocolos de bajo nivel en el código funcional. Sin embargo, se diferencia en que los protocolos semánticos tanto en LFMOC como en ACT deben ser codificados como parte de la funcionalidad que proporcionan los objetos, lo que los hace dependientes del modelo de interacción utilizado.

Otra propuesta interesante es [32], en donde los autores presentan un *framework* denominado DAOF que es capaz de tratar diferentes aspectos mediante intercepción y adaptación de las llamadas a métodos. En DAOF, los aspectos son entidades de primer orden que se pueden componer en tiempo de ejecución utilizando la información almacenada en el middleware. Para tratar un aspecto, éste debe ser encapsulado en una clase y registrado en el middleware, que es responsable de aplicarlos en un orden determinado cada vez que detecta la llamada a un método de un objeto. La propuesta se ilustra mediante el aspecto de coordinación, que es muy similar a LFMOC o ACT. En DAOF, el aspecto de coordinación se implementa mediante un objeto que permite observar y controlar las invocaciones a métodos. Sin embargo, no queda claro si los autores son capaces de tratar la coordinación entre múltiples objetos. Contrariamente a LFMOC y nuestra propuesta, DAOF no proporciona un mecanismo automático para generar los protocolos de bajo nivel.

Como conclusión de los trabajos analizados en este campo específico podemos afirmar que en ninguno de ellos se ha intentado separar el modelo de interacción subyacente de la funcionalidad, pese al hecho de que existan numerosos trabajos en los que se proponen modelos de interacción avanzados mucho más adecuados en determinados contextos que la llamada a métodos [28].

6 Conclusiones

En este artículo hemos explorado la orientación a aspectos, el modelo de interacción entre múltiples participantes y la forma en que la programación de sistemas distribuidos puede beneficiarse de ello. La separación de la funcionalidad y el modelo de interacción favorece la reutilización, mejora la facilidad de comprensión y facilita el mantenimiento y la evolución del software puesto que las clases que son puramente funcionales pueden ser fácilmente reutilizadas en otros contextos en los que existen modelos de interacción específicos más

adecuados que la llamada a métodos. Hemos demostrado, además, que nuestra propuesta es viable adaptando un modelo de interacción conocido y desarrollando un *framework* de clases que permite implementarlo de una forma tan eficiente como si lo hubiéramos hecho artesanalmente; el tiempo de desarrollo, sin embargo, se reduce notablemente.

Hasta donde alcanza nuestro conocimiento, estos resultados son novedosos puesto que ninguna de las referencias de primera línea consultadas intenta separar el modelo de interacción subyacente de otros aspectos. Se asume generalmente que los objetos se comunican mediante llamada a métodos y esto los hace dependientes de este modelo de interacción y dificulta su reutilización en otros contextos en los que son más adecuados otros modelos.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the Workshop on Object-Based Distributed Programming, ECOOP'93*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer-Verlag, 1994.

[2] R.L. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, September 1989.

[3] B. Bauer, J. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[4] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, Redwood City, CA, 1990.

[5] A. Brodsky, D. Brodsky, I. Chan, Y. Coady, J. Pomkoski, and G. Kiczales. Aspect-oriented incremental customization of middleware services. Technical Report TR-2001-06, Department of Computer Science, University of British Columbia, 2001.

[6] D. de Champeaux. Object-oriented analysis and top-down software development. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 360–375. Springer–Verlag, 1991.

[7] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Computing Series. Yourdon Press, Englewood Cliffs, New Jersey, 1990.

[8] R. Corchuelo, J.A. Pérez, and M. Toro. A multiparty coordination aspect language. *ACM Sigplan*, 35(12):24–32, December 2000.

[9] R. Corchuelo, D. Ruiz, M. Toro, J.M. Prieto, and J.L. Arjona. A distributed solution to multiparty interaction. In *Recent Advances in Signal Processing and Communications*, pages 318–323. World Scientific Engineering Society, 1999.

[10] R. Corchuelo, D. Ruiz, M. Toro, and A. Ruiz-Cortés. Implementing multiparty interactions on a network computer. In *Proceedings of the XXV^{th} Euromicro Conference (Workshop on Network Computing)*, Milan (Italy), September 1999. IEEE Press.

[11] R. Corchuelo, A. Ruiz-Cortés, J.R. Mühlbacher, and J. García-Consuegra. Object–oriented business solutions. In *Object-Oriented Technology*, volume 2323 of *Lecture Notes in Computer Science*, chapter 8. Springer Verlag, 2002.

[12] R. Corchuelo, A. Ruiz-Cortés, and R. Wrembel. *Technologies Supporting Business Solutions.* Nova Science Publishers, 2002.

[13] T. Elrad, R.E. Filman, and A. Bader. Aspect–oriented programming. *Communications of the ACM*, 44(10):29–32, 2001.

[14] M. Fayad and R.E. Johnson, editors. *Domain-Specific Application Frameworks: Frameworks Experience by Industry.* John Wiley & Sons, 1999.

[15] M. Fayad and D.C. Schmidt, editors. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley & Sons, 1999.

[16] M. Fayad, D.C. Schmidt, and R.E. Johnson, editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work.* John Wiley & Sons, 1999.

[17] M.E. Fayad and D.C. Schmidt. Lessons learned building reusable OO frameworks for distributed software. *Communications of the ACM*, 40(10):85–87, October 1997.

[18] N. Francez and I. Forman. Synchrony loosening transformations for interacting processes. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of Concurr'91: Theories of concurrency – Unification and extension*, number 527 in Lecture Notes in Computer Science, pages 27–30, Amsterdam, The Netherlands, August 1991. Springer–Verlag.

[19] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.
[20] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison–Wesley Longman, June 1999.
[21] S. Frølund and G. Agha. A language framework for multi-object coordination. In O. Nierstrasz, editor, *Proceedings of the European Conference on Object-oriented Programming ECOOP'93*, pages 346–360, Kaiserslautern, Germany, 1993. Springer–Verlag.
[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W.G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming ECOOP'01*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, 2001.
[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming ECOOP'97*, pages 220–242. Lecture Notes in Computer Science, Springer–Verlag, 1997.
[24] K. Liebherherr. From transience to persistence in object-oriented programming: Patterns and architectures. *ACM Computing Surveys*, 28A(4):39–41, December 1996.
[25] C.V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Xerox Palo Alto Research Center, 1998.
[26] J. Odell, H.V.D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Proceedings of the 22nd International Conference on Software Engineering ISCE'01*, number 1957 in LNCS, pages 121–140. Springer–Verlag, June 2001.
[27] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.
[28] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers*, volume 46. Academic Press, 1998.
[29] J. A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An order–based, distributed algorithm for implementing multiparty interactions. In *Coordination Models and Languages. Proceedings of the 5th International Conference COORDINATION 2002.*, Lecture Notes in Computer Science, pages 250–257, York, United Kingdom, 2002. Springer.
[30] J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. A framework for aspect–oriented multiparty coordination. In *Working Conference on Distributed Applications and Interoperable Systems DAIS'01*, page To appear, Krakow, Poland, 2001. Kluwer Academic Publishers.
[31] J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An enablement detection algorithm for open multiparty interactions. In *Applied Computing 2002. Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 378–384, Madrid, Spain, March 2002. ACM Press.
[32] M. Pinto, L. Fuentes, M.E. Fayad, and J. M. Troya. Separation of coordination in a dynamic aspect-oriented framework. In *Proceedings of the First International Conference on Aspect-Oriented Software Development AOSD'01*, Enschede, The Netherlands, April 2002.
[33] T. Reenskaug, P. Wold, and O.A. Lehne. *Working With Objects. The OOram Software Engineering Method*. Manning Publications Co., August 1995.
[34] A. Ruiz-Cortés, R. Corchuelo, J.A. Pérez, A. Durán, and M. Toro. An aspect–oriented approach based on multiparty interactions to specifying the behaviour of a system. In *Principles, Logics, and Implementations of High-Level Programming Languages PLI'99. Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours*, pages 56–65, Paris, France, 1999.
[35] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley Longman, Reading, Massachussetts, 1999.
[36] A.R. Silva, P. Sousa, and J.A. Marques. Development of distributed applications with separation of concerns. In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference APSEC'95*. IEEE Press, 1995.
[37] J. Viega and D. Evans. Separation of concerns for security. In P. Tarr, A. Finkelstein, W. Harrison, H. Nusibeh, H. Osser, and D. Perry, editors, *Proceedings of the Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE'00*, 2000.
[38] R. Wirfs-Brock and B. Wilkerson. *Designing Object-Oriented Software*. Prentice–Hall, August 1990.
[39] A.F. Zorzo and R.J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. *ACM Sigplan*, 34(10):435–446, October 1999.