CAPÍTULO IV

IMPLEMENTACIÓN EN .NET

Contenidos del capítulo:

IMPLEMENTACIÓN EN .NET

Tras haber presentado las diferentes tecnologías existentes orientadas a aspectos en .NET y el modelo PRISMA, a continuación se describe cómo se ha implementado el modelo sobre la plataforma .NET.

El diseño realizado da soporte a la concurrencia, la comunicación distribuida, la movilidad, la reutilización y la evolución dinámica de código. Éstos son aspectos característicos del modelo arquitectónico PRISMA y proporcionan a sus elementos una gran potencia expresiva. Sirva como ejemplo de la capacidad que brinda el prototipo desarrollado, la propiedad de los componentes de poder agregar o eliminar aspectos, weavings o puertos en tiempo de ejecución sin necesidad de recompilar y, en algunos casos, sin detener la ejecución del componente. Esta propiedad es posible gracias a que los componentes se han implementado de forma totalmente independiente de las partes que lo componen, con lo que además también se favorece la reutilización.

Para la ejecución de las aplicaciones PRISMA se ha desarrollado un *middleware* que da soporte a las características requeridas por el modelo, como la ejecución concurrente de los elementos arquitectónicos, la comunicación distribuida, la movilidad o la evolución dinámica.

A continuación se describirá el diseño de la arquitectura, pasando después a detallar cómo se ha implementado el *middleware*.

4.1 Diseño de la arquitectura

4.1.1 Introducción

En la Figura 21 puede observarse cómo las diferentes tecnologías orientadas a aspectos analizadas en el capítulo anterior actúan sobre los distintos niveles que constituyen el .NET *Framework*. Éstas se pueden dividir en tres grupos [Jack04], separados por la forma de realizar el *weaving* entre código base y aspectos:

- **Modificación del motor de ejecución (el CLR):**

  Son aquellas aproximaciones que introducen mecanismos para extender la funcionalidad del CLR. Por ejemplo, en JAsCo.Net se insertan *traps*[10] a los ensamblados generados, y en Rapier-Loom.Net se modifican las estructuras internas del CLR para que los aspectos puedan ser llamados desde el código base. Los principales inconvenientes de estas aproximaciones son, por una parte, que el modelo de puntos de unión está limitado por los mecanismos utilizados para hacer el *weaving* en el CLR (por ejemplo, en Rapier-Loom.Net, todos los métodos debían ser virtuales, lo que repercute en el rendimiento. Por otra parte, las herramientas de depuración existentes no pueden ser utilizadas ya que el CLR ha visto modificado su comportamiento.

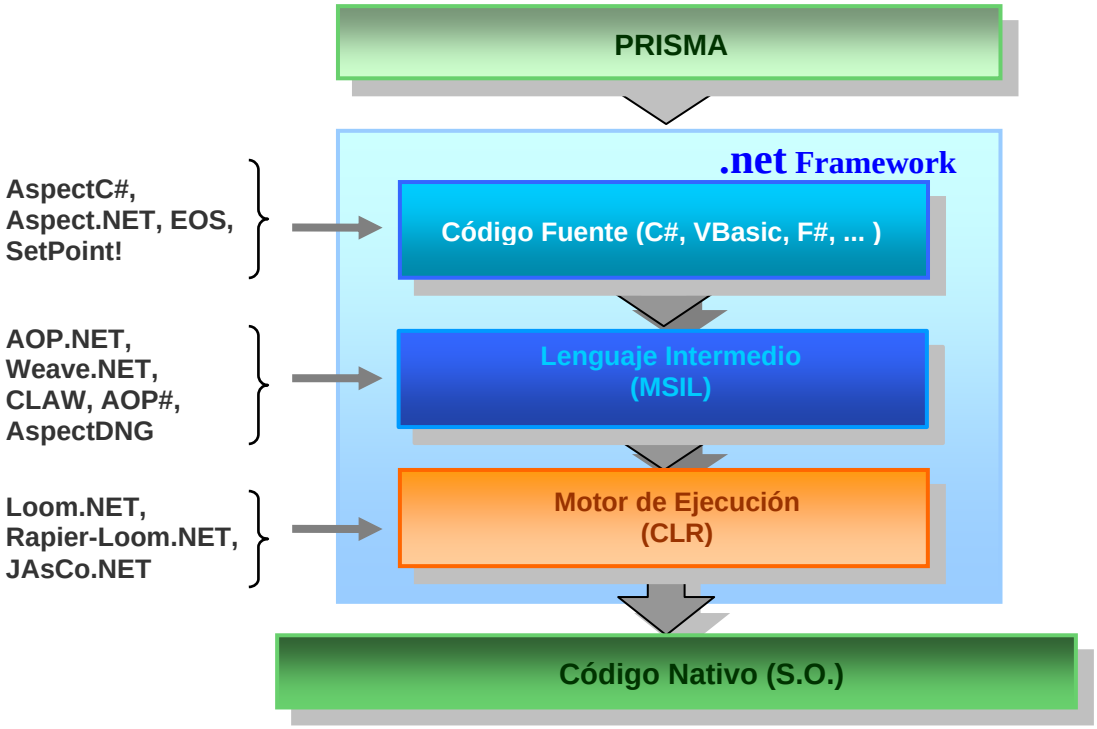

Figura 21 - Aplicación de PRISMA sobre .NET

- **Alteración del código intermedio generado (MSIL):**

  Son las aproximaciones que manipulan de alguna forma el lenguaje intermedio generado para enlazar el código base con los aspectos. Para ello, en la mayoría de los casos se modifica el código intermedio de los ensamblados para que sus métodos sean interceptados en tiempo de ejecución y se procesen los aspectos. Aunque se mantiene la independencia del lenguaje utilizado, el principal inconveniente es que al igual que en el caso anterior, tampoco es posible depurar el código fuente.

[10] En este contexto, una *trap* (trampa) es una invocación hacia un método externo.

- **Extensión del lenguaje CLS/CTS:**

  Se incluyen en este grupo aquellas aproximaciones que extienden el CLS/CTS para permitir la unión de aspectos en tiempo de compilación. El problema que presentan es que los compiladores desarrollados deben actualizarse al mismo ritmo que la evolución de los lenguajes .NET que extienden, y además, se pierde la independencia del lenguaje.

Los tres grupos descritos tienen el inconveniente adicional de crear una dependencia con la plataforma .NET, con lo que futuras versiones provocarán una actualización de dichas aproximaciones para mantener la compatibilidad.

En cambio, PRISMA actúa a un nivel superior, ya que utiliza un lenguaje de descripción de arquitecturas independiente de cualquier plataforma de desarrollo, lo que le permite incorporar funcionalidad adicional y un mayor nivel de abstracción. Por otro lado, la implementación de PRISMA se ha realizado basándose en los lenguajes estándar definidos, sin añadir ninguna extensión a la plataforma de desarrollo, con el objetivo de no tener que adaptar la implementación a nuevas versiones del CLR, MSIL, etc.

El proceso de crear un modelo PRISMA y ejecutarlo pasa por una serie de etapas (ver Figura 22). En primer lugar, a través de una herramienta visual, el analista diseña y especifica el modelo arquitectónico. En la segunda etapa, a partir del modelo arquitectónico se genera su código intermedio, dependiente de la plataforma de desarrollo, en nuestro caso C#. Por último, el código generado es puesto en ejecución por el *middleware PRISMA*, el cual implementa el modelo PRISMA y proporciona los servicios de distribución y evolución requeridos por el modelo.

En este proyecto se ha construido en C# un prototipo de dicho *middleware* y se han detectado las correspondencias entre el modelo PRISMA y la implementación, lo que permitirá en un futuro construir el compilador.

4.1.2 El Middleware PRISMA

Por una parte, el *Middleware* ofrece una serie de clases que implementan la funcionalidad básica de los tipos PRISMA. Mediante la extensión de estas clases es posible la implementación de las aplicaciones PRISMA.

Por otra parte, el *Middleware* ofrece servicios tanto a componentes, conectores y sistemas del modelo PRISMA que están ejecutándose localmente, como a otros *Middleware*s PRISMA en ejecución en otras máquinas. Algunos de los servicios ofrecidos son la carga de componentes, creación de los hilos de ejecución, gestionar las listas de componentes existentes localmente, etc.

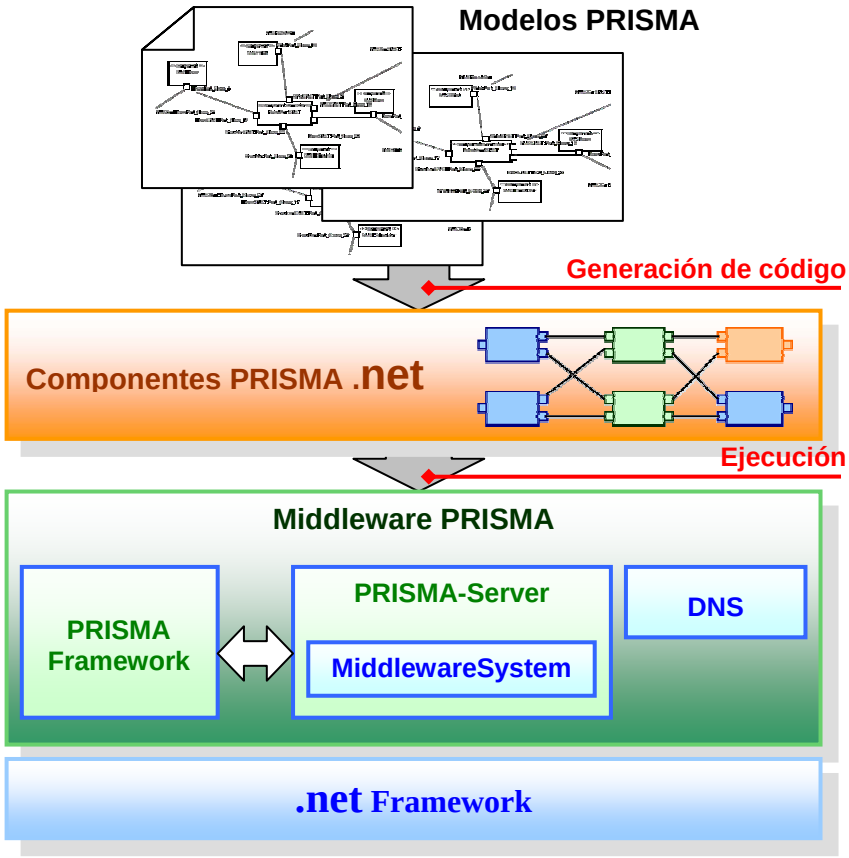

Figura 22 - Arquitectura del Middleware PRISMA

El *middleware* (ver Figura 22) es una capa adicional que se ejecuta sobre la plataforma .NET, formada a su vez por tres módulos:

- **PRISMA Server**: Módulo que contiene el núcleo del *Middleware*, implementado en la clase *MiddlewareSystem*, y proporciona los servicios de gestión, distribución y evolución de los componentes PRISMA.
- **PRISMA Framework:** Consola de administración del *Middleware* que ofrece al usuario los servicios proporcionados por el núcleo, y muestra los mensajes de estado generados por el *middleware*.
- **Servidor DNS**: Servicio de nombres para localizar la ubicación de los distintos componentes del sistema.

Los tres módulos son independientes entre sí, para que en caso de fallar uno de ellos, el sistema pueda recuperarse fácilmente sin más que volver a cargar el módulo. Sin embargo, la tolerancia a fallos no ha sido incorporada, y queda como una tarea futura para próximos prototipos.

Un modelo PRISMA en ejecución puede ser distribuido, tal como se contempla en el lenguaje de descripción de arquitecturas. Esto implica que en cada nodo donde se quiera ejecutar PRISMA debe estar instalado y cargado el *middleware*, ya que además de gestionar los componentes PRISMA también proporciona los servicios de movilidad, replicación, acceso remoto, etc. La arquitectura de dichos nodos es totalmente descentralizada.

Los distintos *middleware* se encargan de gestionar los elementos que están ejecutándose en su nodo y de establecer las comunicaciones con sus *middlewares* vecinos. Los elementos compilados del modelo PRISMA se encuentran distribuidos por los distintos nodos, de forma que si uno de ellos falla, el sistema formado por el conjunto de *middlewares* puede redistribuirse las tareas para mantener estable al sistema.

El prototipo que se describe en este proyecto, aunque diseñado para soportar una arquitectura descentralizada, utiliza un servicio de nombres centralizado para albergar la ubicación de todos los elementos arquitectónicos: es el módulo DNS. No obstante, al estar diseñado como un elemento independiente, la sustitución futura de dicho módulo por una estrategia descentralizada no supondrá un gran impacto sobre el prototipo implementado. Según el ADL de PRISMA mostrado en el capítulo anterior, el servicio de nombres del modelo se encontrará distribuido en los *attachments* y *bindings*, por lo que no será necesario el módulo DNS.

Dada una aplicación PRISMA, se pueden distinguir tres clases de comunicaciones:

- las realizadas por parte de los elementos arquitectónicos de la aplicación al *Middleware*, en las que se solicitan servicios de movilidad o de replicación.
- las comunicaciones entre los distintos elementos arquitectónicos, según lo requieran las especificaciones del modelo (solicitud de servicios unos a otros).
- las realizadas entre los distintos *Middlewares* para averiguar las localizaciones de los distintos elementos, mover elementos, transferir tipos, evolucionar tipos y arquitecturas, actualizar el estado, etc.

Debido a la naturaleza distribuida del modelo, la implementación de las comunicaciones se ha realizado con la tecnología .Net Remoting.

Por otra parte, el PRISMA *Framework* es un módulo opcional, únicamente necesitará cargarse en un nodo desde el que se desee administrar la configuración del sistema. Entre sus funciones están las de cargar modelos PRISMA, permitir la evolución del modelo, comprobar el estado de los distintos elementos, etc.

4.2 Implementación del Middleware

Una vez vista la arquitectura del *middleware PRISMA*, a continuación se describe cómo se han implementado los distintos conceptos PRISMA en .NET. En primer lugar se presentan de forma general los mecanismos que permiten ejecutar los servicios de forma asíncrona, gracias a la clase *AsyncResult*. En segundo lugar se presentan las interfaces y los aspectos,

describiendo las correspondencias entre PRISMA y .NET, cómo se definen los distintos tipos de aspectos y su modelo de ejecución. En tercer lugar se presenta la implementación de los componentes, su modelo de ejecución y cómo gestionan los aspectos, weavings y puertos. En cuarto lugar, se presenta la implementación de los sistemas, que permiten la creación de componentes complejos, y de los attachments y bindings, los cuales establecen el modelo de comunicación entre componentes, conectores y sistemas. Por último, se presenta la implementación de la clase *MiddlewareSystem* y se describen los distintos tipos de servicios ofrecidos.

Los diferentes tipos PRISMA se han implementado en una serie de clases, agrupadas según su funcionalidad en diferentes espacios de nombres, mostrados en la Figura 23. Estas clases proporcionan el comportamiento genérico de los distintos tipos definidos en PRISMA. La implementación de cada tipo específico definido mediante el ADL de PRISMA, es generada por el compilador de acuerdo a las plantillas de código que en este capítulo se presentan. Dichas plantillas únicamente definen la estructura básica de cada tipo, el comportamiento es heredado de las clases genéricas agrupadas en el espacio de nombres de PRISMA.

En los apartados siguientes se describen estas clases y las plantillas de código asociadas. Para cada tipo PRISMA se muestran los diagramas de clases correspondientes a la etapa de diseño. También es preciso indicar que, cuando se mencione al *middleware*, se hace referencia a la clase *MiddlewareSystem*, que proporciona toda la funcionalidad de bajo nivel.

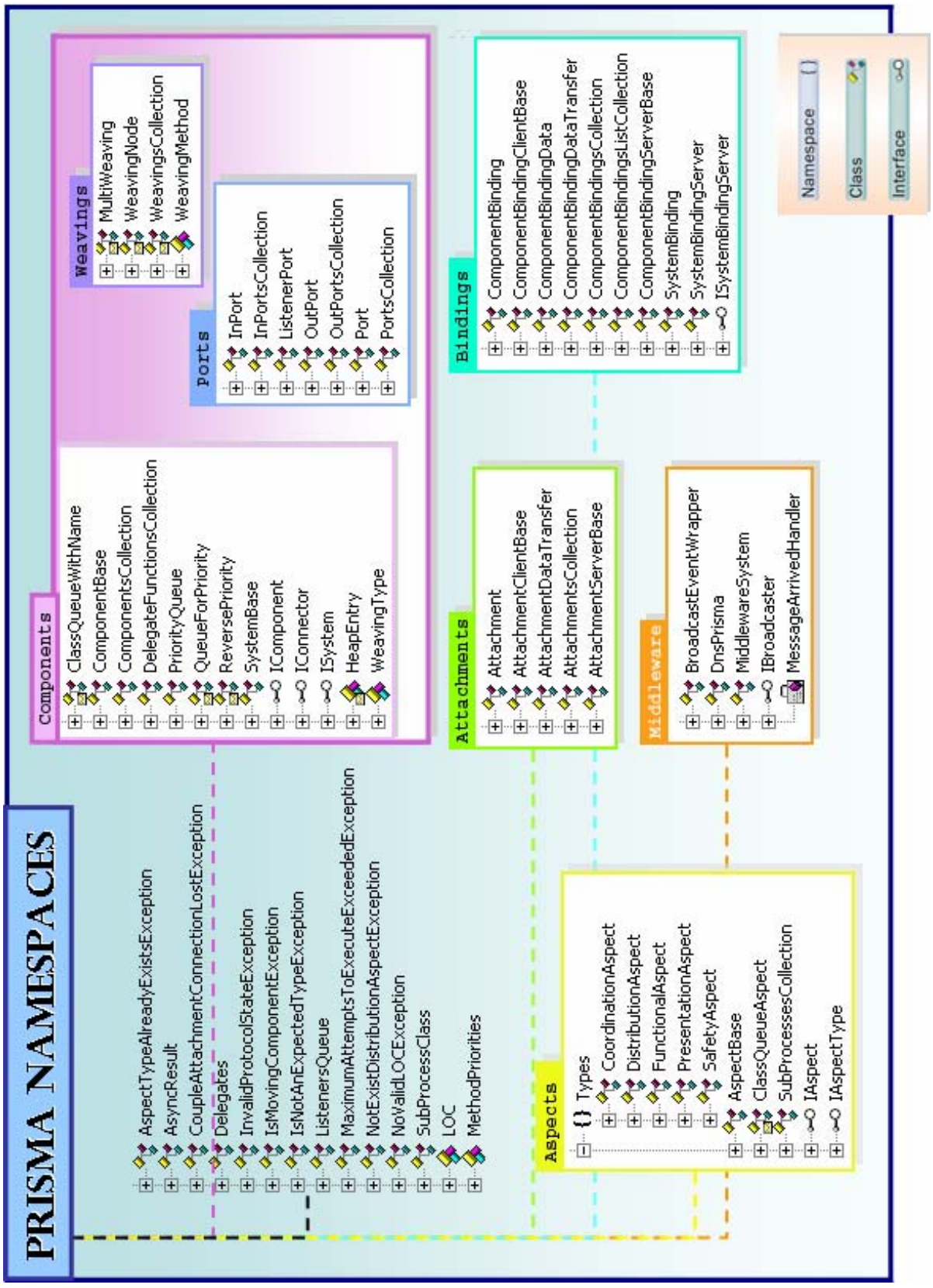

Figura 23 - Espacio de nombres de la implementación de PRISMA

4.2.1 La clase *AsyncResult*

Antes de pasar a los tipos PRISMA, es necesario introducir la clase *AsyncResult*, pues es un tipo fundamental que ha permitido implementar la concurrencia y las llamadas asíncronas entre los diferentes procesos.

En .NET existen mecanismos para realizar llamadas asíncronas, a través de las clases *Delegate* y *AsyncResult*. El cliente de un servicio asíncrono debe crear un delegado para el servicio requerido y ejecutarlo mediante el servicio *BeginInvoke*, obteniendo una estructura *AsyncResult* que le permitirá conocer la finalización del servicio y obtener los resultados. Si el cliente no desea procesar los resultados y desea continuar su proceso interno, necesariamente siempre deberá ejecutar el servicio de terminación *AsyncResult.EndInvoke*. Por otra parte, si dicha estructura se genera en una máquina remota, no puede ser accedida remotamente, pues tiene fuertes dependencias de implementación con la máquina sobre la que se ha generado. Además, la asincronía proporcionada es gestionada por la plataforma de desarrollo, lo cual es un inconveniente para la implementación del modelo PRISMA, ya que éste requiere una serie de mecanismos adicionales para gestionar las peticiones recibidas, establecer un orden de ejecución, redireccionar servicios a otros elementos, etc.

Por ello, se ha creado la clase *AsyncResult* (que no debe confundirse con la proporcionada por la plataforma .NET) cuya finalidad es la de servir como nexo de unión entre el cliente y el proveedor de un determinado servicio. Esta clase permite implementar el comportamiento asíncrono de forma transparente para el cliente, ocultando los detalles específicos de la plataforma de desarrollo (como el uso de delegados), y puede ser extendida para añadir funcionalidad adicional. Por ejemplo, en una primera versión, el tratamiento de excepciones no estaba contemplado. En la versión actual se ha implementado un mecanismo sencillo que permite propagar al cliente de un servicio las excepciones generadas remotamente.

La asincronía en la ejecución de un servicio es proporcionada a través de varios mecanismos, siendo esta clase la estructura que permite posteriormente comprobar si la petición ha sido procesada. El modelo de ejecución mediante el cual se implementa la asincronía, cuyos detalles pueden variar en cada caso concreto, se presenta a continuación. Para ello, se ilustra utilizando un ejemplo concreto.

De forma general, un cliente solicita un servicio a un proveedor de servicios (servidor). Todos los servicios susceptibles de ser invocados asíncronamente devolverán siempre una estructura *AsyncResult*, mediante la cual podrá comprobarse la finalización de dichos servicios o acceder a los resultados devueltos. Por ejemplo, la definición del servicio *Balance* publicada por el servidor sería la siguiente:

```
AsyncResult Balance(ref decimal money);
```

Nótese que este servicio tiene un parámetro de salida en el cual se depositará el saldo de una cuenta, y que devuelve un objeto *AsyncResult*. Debido a que el servidor puede encontrarse ejecutando un servicio solicitado previamente, debe incorporar un mecanismo que le permita almacenar las nuevas peticiones de servicio recibidas para su posterior tratamiento. Por esta razón, por cada petición de servicio recibida, el servidor crea un delegado para ejecutar posteriormente el servicio solicitado y una instancia de la clase *AsyncResult* asociada a dicho servicio. Esta instancia, junto con el delegado, son almacenados en la cola del servidor para ser procesados posteriormente. Por ejemplo, el código correspondiente al cuerpo del método *Balance* implementado en el servidor sería el siguiente:

```
public AsyncResult Balance(ref decimal money) {
   // Creación del delegado hacia el método privado con la
   // implementación de Balance.
   balanceDelegate = new _BalanceDelegate(this._Balance);
   // Creación del vector de argumentos
   object[] args = new object {money};

   // Creación del objeto AsyncResult
   AsyncResult result = new PRISMA.AsyncResult(args.Length);
   // Creación del nodo de la cola
   queueNode newNode = new queueNode(balanceDelegate, args, result);
   // Se encola la petición en la cola de servicios
   queue.Enqueue(newNode);
   // Se devuelve el objeto AsyncResult
   return result;
}
```

La creación de un objeto *AsyncResult* se realiza proporcionando la longitud del vector de parámetros del método al cual va asociado, ya que internamente se crea un vector de dicho tamaño en el que se almacenarán los parámetros devueltos por dicho método. Al cliente le retorna la referencia al objeto *AsyncResult* creado, mediante el cual éste podrá comprobar si su petición de servicio ha sido procesada. A partir de este momento, el cliente podrá continuar realizando sus tareas, hasta que el servidor retorne los resultados del servicio. Esto puede comprobarse mediante la invocación del método *HasResult*(), que indica si el objeto ya tiene los valores devueltos por el servicio asociado. Continuando con el ejemplo anterior, el código correspondiente para que el cliente pueda comprobar si el método *Balance* ya se ha procesado, sería el siguiente:

```
result = Balance(ref money);
// Realización de otras tareas ...
// ...
// Ahora espera hasta obtener el resultado del método Balance.
while (!result.HasResult()) System.Threading.Thread.Sleep(100);
```

Cuando el servidor procese la petición almacenada en la cola, lo que implica ejecutar el delegado y con ello invocar el servicio asociado, depositará los resultados obtenidos en el objeto *asyncResult* creado anteriormente, y al cual el cliente tiene acceso. Para ello, se invoca el método *SetParameters*

proporcionando como argumento un vector que corresponde a los parámetros devueltos por la ejecución del servicio requerido. El código correspondiente al procesado del servicio por parte del servidor es el siguiente:

```
// Se desencola el siguiente servicio de la cola
queueNode newNode = (queueNode) queue.Dequeue();
// Se ejecuta el delegado almacenado en dicho nodo, proporcionándole
// los parámetros proporcionados cuando se creó
newNode.delegate.DynamicInvoke(newNode.Parameters);
// Se actualiza el objeto AsyncResult
newNode.Result.SetParameters(newNode.Parameters);
```

Finalmente, en la parte del cliente, cuando el método *HasResult* devuelva verdadero, éste podrá obtener los resultados del servicio solicitado a través de la invocación del método *GetParameters*, que devolverá el vector de parámetros devuelto por el servicio.

```
// Ya ha finalizado el servicio, se obtienen los resultados
object[] args = result.GetParameters();
```

Por otro lado, las peticiones de servicio recibidas pueden ser reenviadas a otros elementos, que pueden ser locales o remotos, de forma totalmente transparente para el cliente. Esto es posible encadenando entre sí una serie de objetos *asyncResult* de tal forma que la actualización de sus resultados se propague desde un extremo de la cadena, que puede residir en otra máquina distinta, hasta el otro extremo, el cliente del servicio. Para ello, se ha definido un atributo *derivedResult*, que puede contener una referencia a otro objeto *asyncResult*. Cuando es invocado el método *HasResult* o *GetParameters*, se comprueba previamente si dicho atributo contiene una referencia válida, en cuyo caso se invoca recursivamente hasta obtener el resultado. Como la clase *AsyncResult* hereda de *MarshalByRefObject*, puede ser accedida remotamente, lo que permite que los resultados de un servicio ejecutado en otra máquina sean accesibles remotamente a través del objeto *asyncResult*.

En la Figura 24 se muestra el diagrama de secuencia correspondiente para el ejemplo mostrado anteriormente, en el que un objeto *Client* solicita un servicio *Balance* a otro objeto *Server*. Éste, cuando procesa la petición recibida, la reenvía a otro servidor remoto (*RemoteServer*) que le devuelve una referencia al objeto *AsyncResult* creado en aquella máquina. Para que el cliente pueda obtener posteriormente los resultados calculados en la otra máquina, el objeto *Server* invoca el método *SetDerivedParameters,* pasando como parámetro la referencia del objeto *asyncResult* remoto. De esta forma, cuando el cliente solicita los resultados, éstos son devueltos desde la ubicación remota.

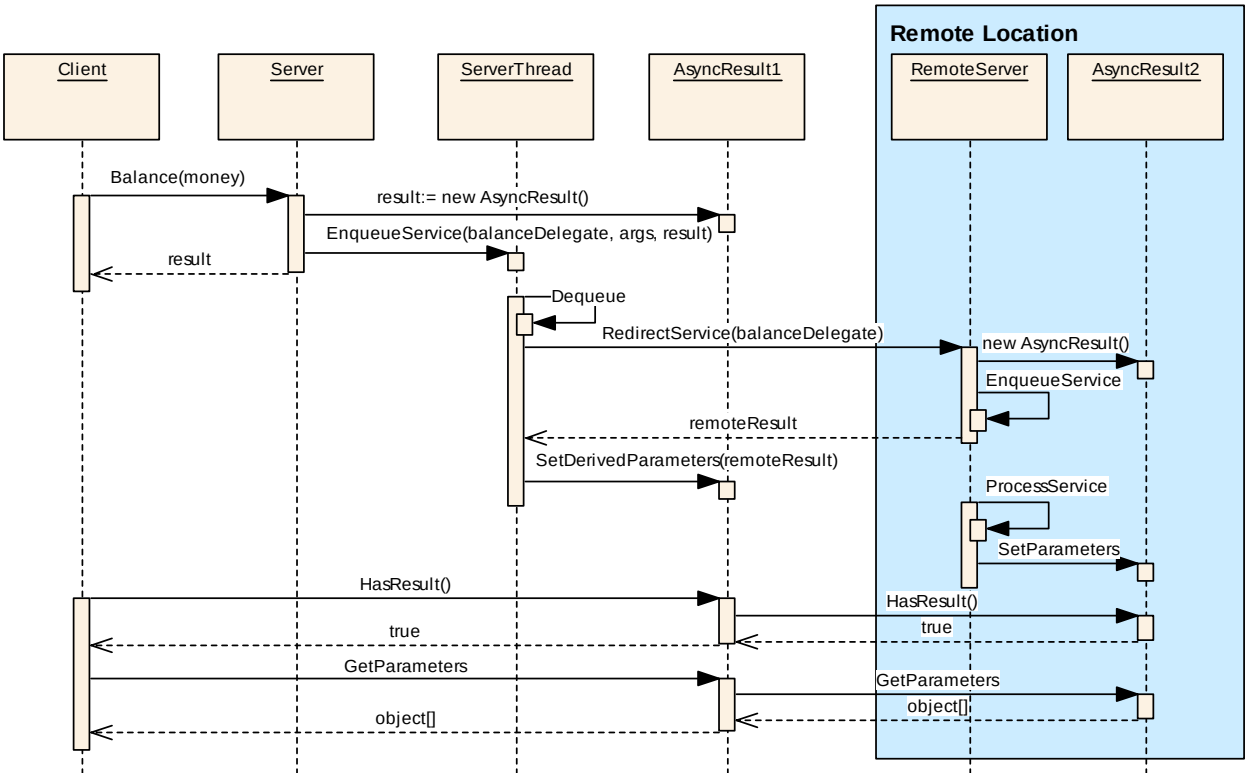

Figura 24 - Esquema de funcionamiento de la clase AsyncResult

En la Figura 25 se muestran los atributos y servicios que la clase *AsyncResult* define:

- **parameters**: Vector de objetos en el que se almacenan los parámetros devueltos por la ejecución del servicio asociado.
- **exception**, **ThrowExceptionToClient()**: el atributo privado *exception* permite almacenar una excepción para ser retransmitida al cliente, cuyo valor es establecido a través de la invocación del método *ThrowExceptionToClient*. Cuando el servidor detecta una excepción provocada por el servicio que estaba en ejecución, almacena dicha excepción en el objeto *asyncResult* mediante la invocación de este método, que establece el atributo *hasResults*=true, para que el cliente sepa que la ejecución del servicio ha finalizado. Cuando el cliente invoque el método *GetParameters,* la excepción será relanzada, pero esta vez en el ámbito de ejecución del cliente.
- **derivedResult**, **setDerivedParameters()**: contiene una referencia a otro objeto *asyncResult*, permitiendo que un resultado obtenido remotamente se transmita de forma recursiva hasta la instancia a la cual el cliente tiene acceso. El método permite inicializar este atributo con otro objeto *asyncResult* proporcionado como parámetro.
- **hasResults** y **HasResult()**: el atributo *hasResults* es inicializado a *true* cuando los parámetros son almacenados en el atributo *parameters*. El método *HasResult* devuelve el valor de este atributo,

pero comprueba previamente si existe otro objeto *asyncResult* asociado al atributo *derivedResult*.

- **SetParameters()**, **GetParameters()**: el método *SetParameters* permite establecer el valor del atributo *parameters*, mientras que el método *GetParameters* permite obtener el vector almacenado en dicho atributo. La invocación del primer método conlleva también que el atributo *hasResults* sea puesto a verdadero, para que el cliente sepa que la ejecución del servicio ha finalizado. La invocación del método *GetParameters* comprueba previamente si hay otro objeto *asyncResult* asociado a esta instancia. En caso afirmativo, devolverá el resultado de la invocación *derivedResult.GetParameters()*.

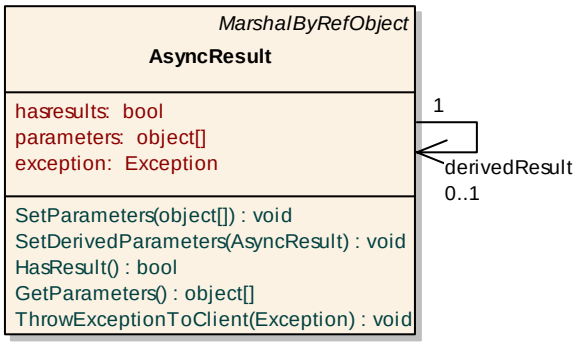

Figura 25 - La clase AsyncResult

De este modo, para incorporar la ejecución asíncrona de los servicios ofrecidos por los tipos PRISMA, y permitir el procesamiento concurrente de múltiples servicios, se establece que todo servicio ofrecido por los tipos PRISMA devuelva como resultado un objeto de tipo *AsyncResult*. Esto es posible gracias a que en la especificación PRISMA todos los servicios se comportan como procedimientos, es decir, devuelven sus resultados a través de parámetros de salida, por lo que puede utilizarse el valor de retorno para devolver el objeto correspondiente *AsyncResult*.

4.2.2 Interfaces

Las interfaces PRISMA se corresponden con las interfaces de .NET. Continuando con el ejemplo de la cuenta bancaria presentado en el capítulo anterior (ver sección 3.4.1.1), seguidamente se muestra cómo se han implementado en C# las interfaces de dicho sistema de información:

```
// Definición de la interfaz ICreditCardTransactions
public interface ICreditCardTransactions {
   AsyncResult Withdrawal(decimal quantity, ref decimal money);
   AsyncResult Balance(ref decimal money);
   AsyncResult ChangeAddress(string newAdd);
}

// Definición de la interfaz IMobility
public interface IMobility {
```

```
   AsyncResult Move(LOC newLoc);
}
```

Los parámetros de salida en .NET se identifican por aquellos que llevan la palabra reservada **ref**. Nótese que todos los servicios devuelven un objeto del tipo *AsyncResult*, para permitir el procesamiento asíncrono de los servicios y la ejecución concurrente, como se ha descrito en el apartado anterior.

4.2.3 Aspectos

La implementación de los aspectos PRISMA se compone, por una parte, de la definición del comportamiento especificado para un aspecto en el ADL PRISMA, y por otra parte, de la infraestructura de ejecución que permite la concurrencia de aspectos y la integración con los demás tipos PRISMA.

4.2.3.1 Correspondencia entre PRISMA y la implementación

Para describir la correspondencia entre un aspecto PRISMA y la implementación en .NET, se utilizará como ejemplo el aspecto funcional *BankInteraction*, cuya especificación se mostró en la sección 3.4.1.2, y el código C# asociado.

Un aspecto PRISMA se corresponde con una clase C# cuyo nombre se corresponde con el definido en la especificación (*BankInteraction*), que hereda de otra clase que representa al tipo del aspecto (*FunctionalAspect*) y que implementa la interfaz definida en la especificación (*ICreditCardTransactions*).

```
// Definición del Aspecto Funcional "BankInteraction"
public class BankInteraction : FunctionalAspect, ICreditCardTransactions { }
```

Los atributos de la especificación se corresponden con variables privadas C#, y mediante una enumeración *ProtocolState* se definen los estados posibles de la ejecución de un aspecto. El estado actual en el que se encuentra un aspecto se almacena también en una variable privada.

```
// Definición de atributos PRISMA
int numberId;
string address;
decimal money;

// Definición de los estados posibles del aspecto
enum protocolStates {
   BANKINTERACTION,
   TRANSACTION
}

// Almacena el estado actual del aspecto
protocolStates estado;
```

Un *played_role* se corresponde con la clase *SubprocessClass* (ver Figura 26), que almacena la prioridad y el nombre de los servicios asociados a cada

played_role. Los distintos objetos *SubprocessClass* (los *played_roles* definidos) se almacenan en la clase *SubprocessCollection*, que a su vez está agregada al aspecto a través de la propiedad *subProcessesList*. Con el siguiente fragmento de código se entenderá mejor el proceso de definir y agregar un nuevo *played_role*:

```
// Creación de un nuevo played_role "SERVIDOR"
SubProcessClass subProcessICredit = new SubProcessClass("SERVIDOR");
// Se añaden los métodos que lo forman, junto a su prioridad relativa
subProcessICredit.AddMethod("Withdrawal", 10);
subProcessICredit.AddMethod("Balance", 10);
subProcessICredit.AddMethod("ChangeAddress", 10);
subProcessICredit.AddMethod("Transfer", 10);

// Se añade el nuevo played_role al aspecto
this.subProcessesList.Add(subProcessICredit);
```

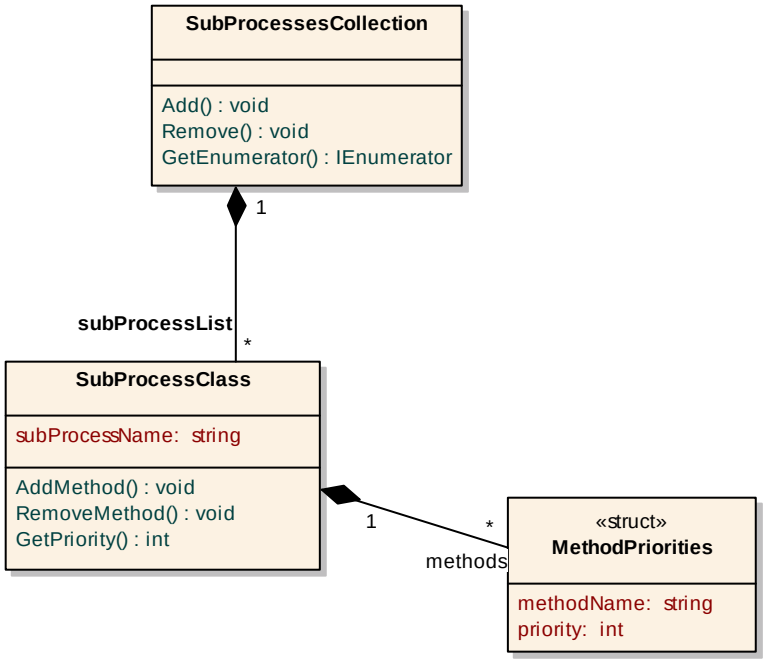

Figura 26 - Clases SubprocessCollection y SubProcessClass

Los servicios PRISMA se corresponden con métodos privados de la clase que implementa al aspecto, en este caso *BankInteraction*. La estructura de un servicio sigue el siguiente patrón:

1. Se comprueba si el estado actual es válido para ejecutar dicho servicio, sino se lanza una excepción.
2. Se comprueba que se cumplan las precondiciones asociadas a dicho servicio.
3. Se comprueba si el estado previo de la valuación se cumple.
4. Se lleva a cabo la funcionalidad asociada.
5. Se comprueba si se debe ejecutar algún disparador.
6. Se asigna el nuevo estado alcanzado a la variable local del estado.

El código correspondiente para definir el servicio *Withdrawal* sería el siguiente:

```
private AsyncResult _Withdrawal(decimal quantity, ref decimal money) {
   // Comprobacion de que el protocolo actual es el correcto
   if (estado != protocolStates.TRANSACTION) {
      throw new InvalidProtocolStateException(this.aspectName, "Withdrawal");
   }

   // Comprobacion de las precondiciones asociadas
   if (quantity > this.money ) {
      throw new Exception("There aren't enough money to withdraw");
   }

   // Comprobación del estado anterior a la valuación
   //   NO APLICABLE

   // Ejecución de la funcionalidad asociada
   this.money -= quantity;
   money = this.money;

   // Comprobación de la activación de Triggers
   //   NO APLICABLE

   // Nuevo estado alcanzado
   estado = protocolStates.TRANSACTION;
   return null;
}
```

Sin embargo, esta implementación no soporta la evolución de la especificación de un aspecto. En el prototipo actual, la evolución de los aspectos se realiza mediante reconfiguración dinámica, ya que los componentes sí que permiten la sustitución de unos aspectos por otros. De esta forma, para cambiar la especificación de un aspecto en ejecución, deberá primero modificarse la especificación PRISMA del aspecto, compilarse y sustituir al aspecto en ejecución.

4.2.3.2 Tipos de Aspectos

Los tipos de aspecto en PRISMA no están limitados, sino que el desarrollador puede definir nuevos tipos en función de los requisitos del modelo a desarrollar. Por ello, la interfaz *IAspectType* define los atributos y servicios básicos que todo tipo de aspecto debe implementar:

- **AspectType**: Atributo que a través de la reflexión de .NET debe almacenar el objeto *Type* correspondiente al tipo del aspecto, con el objetivo de poder identificar fácilmente el tipo de un aspecto sin tener que recorrer toda la jerarquía de herencia.
- **IsSameTypeAs**: Servicio que comprueba si el aspecto que se le pasa como parámetro es del mismo tipo que el aspecto que lo implementa.
- **CheckPreconditions**: Servicio que define las condiciones que debe cumplir el componente que va a incorporar el tipo del aspecto. Por ejemplo, el aspecto funcional requiere que sea agregado a un componente, mientras que el de coordinación sólo puede agregarse a conectores.

- **ToString**: este servicio únicamente es utilizado para poder obtener una cadena que identifique el tipo del aspecto.

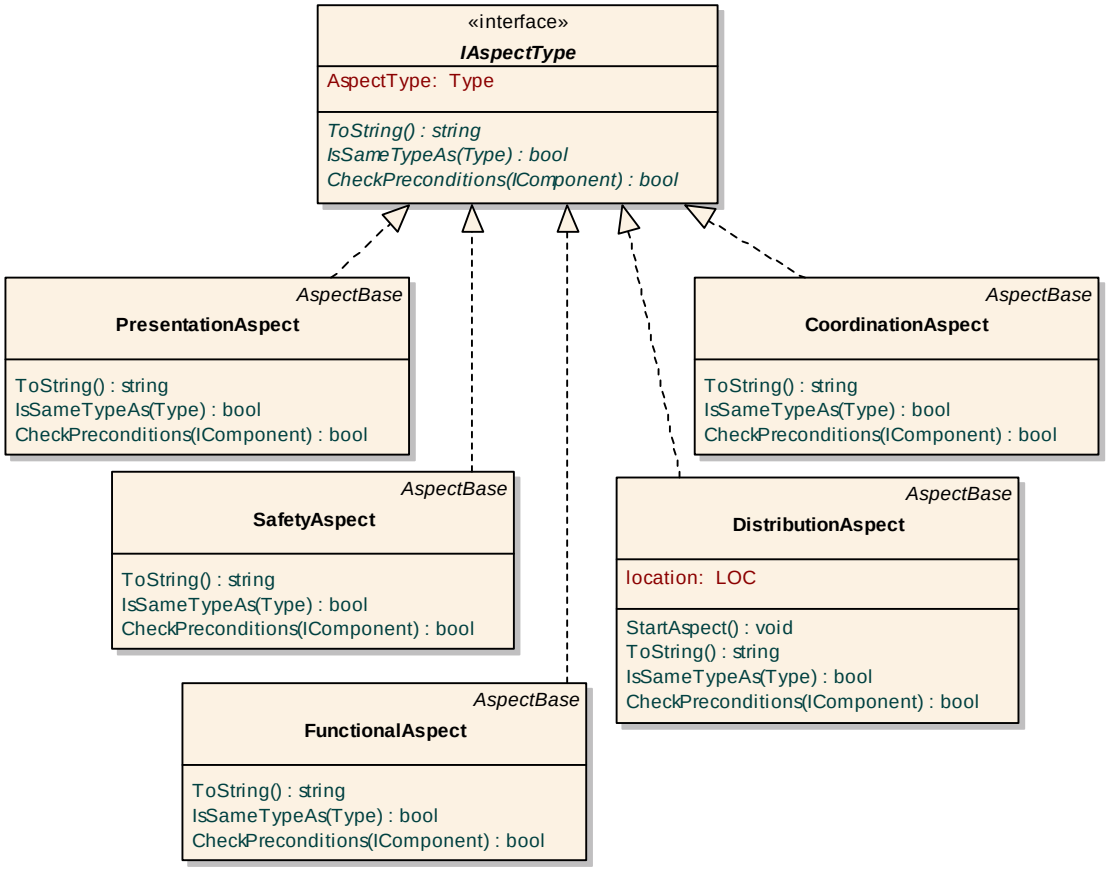

Figura 27 - Interfaz IAspectType y definiciones de tipos de aspectos

Los distintos tipos de aspectos (*FunctionalAspect*, *DistributionAspect*, etc.) además de implementar la interfaz anterior para identificarse como un tipo de aspecto válido en el sistema, deben heredar su comportamiento de la clase *AspectBase*, que como se verá en el siguiente apartado, define el modelo de ejecución común a todos los aspectos. Por otra parte, si los tipos de aspectos requieren atributos adicionales específicos, pueden ser agregados a este nivel e inicializarse con valores por defecto sobreescribiendo el método *StartAspect*(), que es el encargado de iniciar la ejecución de un aspecto. A continuación se muestra cómo se ha implementado el aspecto funcional, con objetivo de ilustrar estos conceptos:

```
public class FunctionalAspect : AspectBase, IAspectType {

   // Constructor del tipo "aspecto funcional"
   // El parámetro "aspectName" es el nombre del aspecto y es requerido
   // por el constructor de AspectBase
   public FunctionalAspect(string aspectName) : base(aspectName) {}

   // Devuelve el nombre del tipo del aspecto
   public override string ToString() { return "FunctionalAspect"; }

   // Devuelve el tipo del aspecto
   public Type AspectType { get { return typeof(FunctionalAspect); } }

   // Comprueba si dos aspectos son del mismo tipo o no
```

```
   public bool IsSameTypeAs(Type obj) { return (this.AspectType.Equals(obj)); }

   // Define las condiciones que debe cumplir el componente que va a incorporar
   // este tipo de aspecto. El parámetro "comp" es una referencia al componente
   // que va a incorporar este tipo de aspecto.
   // Devuelve verdadero si cumple las precondiciones definidas.
   public bool CheckPreconditions(IComponent comp) {
      // Comprobamos que el componente NO implementa la interfaz IConnector
      if (comp is IConnector)
         throw new Exception("FunctionalAspect can't be added to a Connector. "
            + "It must be a Component");
      return true;
   }
}
```

4.2.3.3 Modelo de ejecución

El modelo de ejecución de los aspectos establece cómo se comportan los aspectos y cómo interactúan con los demás tipos del modelo. Como se ha visto en los apartados anteriores, la funcionalidad básica está definida en la clase *AspectBase*, que a su vez implementa la interfaz *IAspect*.

Esta interfaz define los atributos y servicios básicos que todo aspecto debe proporcionar. Permite, mediante el polimorfismo, que el *middleware* pueda tratar de la misma forma a todos los aspectos, o que el componente sepa que se le están agregando aspectos y no otros elementos. Estos atributos y servicios se describen a continuación:

- **aspectName**: es el nombre identificativo del aspecto.
- **aspectThread**: para el modelo de concurrencia es necesario poder obtener el hilo de ejecución asociado a un aspecto, y por ello es necesario un atributo que devuelva una referencia a dicho *thread*.
- **middlewareServer**: es necesaria una referencia al *middleware PRISMA* para aquellas ocasiones en que el aspecto pueda requerir sus servicios, como para la movilidad. Si el componente que posee el aspecto se mueve a otra máquina, dicha referencia debe actualizarse con la referencia del nuevo *middleware*.
- **componentLink**: el aspecto también puede requerir acceder al contexto del componente en el que está alojado, y lo hará a través de esta referencia.
- los servicios **startAspect, stopAspect** y **abortAspect**, que permiten iniciar, parar temporalmente (i.e. para moverse) o detener totalmente el aspecto, respectivamente.

Puesto que los aspectos se ejecutan concurrentemente, el modelo de ejecución debe proporcionar mecanismos para garantizar el procesado de la petición de un servicio mientras el aspecto está ocupado procesando otro, sin bloquear la ejecución del cliente que solicita el servicio. Para dar soporte a esta concurrencia, se ha implementado un sistema de colas en el cual se almacenan las distintas peticiones a medida que son recibidas, mientras el hilo de ejecución del aspecto las va procesando. Este mecanismo es

proporcionado por la clase *AspectBase* (ver Figura 28), de la cual todos los tipos de aspecto heredan su comportamiento.

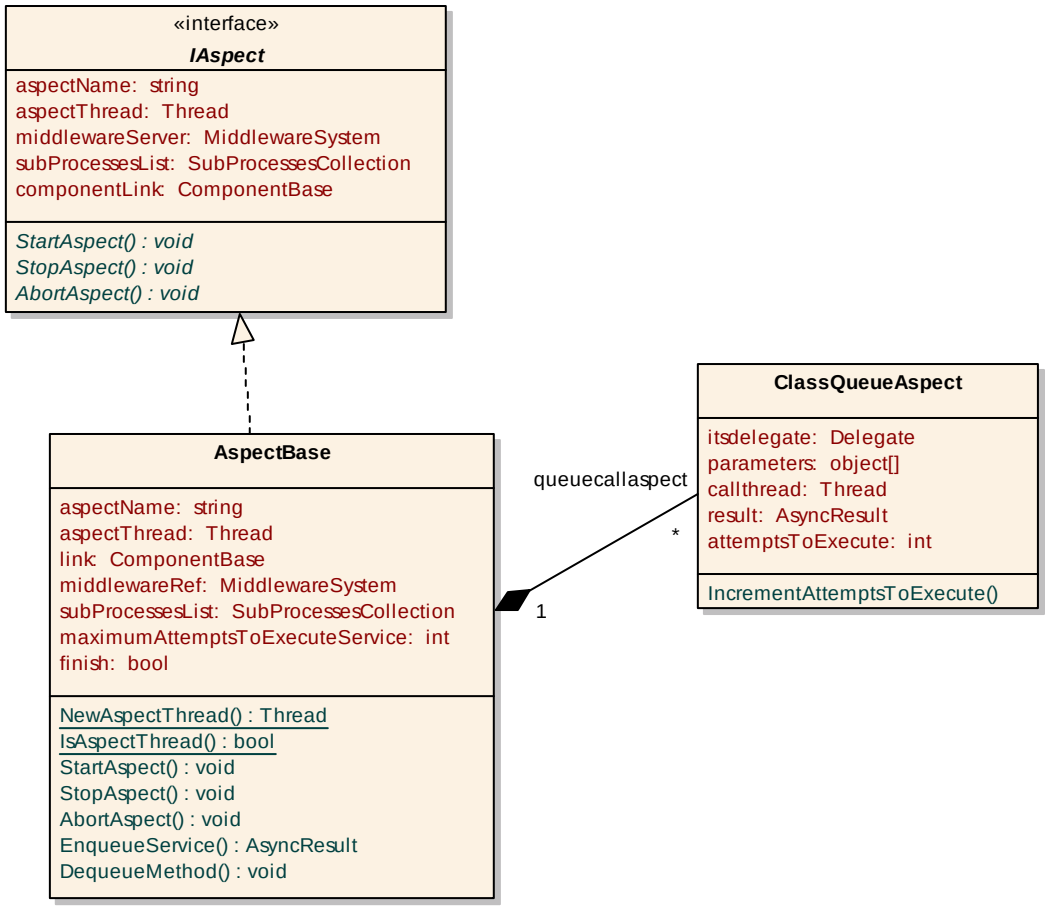

Figura 28 - Interfaz IAspect y clase AspectBase

La cola del aspecto está sincronizada para tolerar el acceso concurrente de múltiples hilos, pues cada petición de servicio puede estar originada por un hilo distinto (por ejemplo, como se verá más adelante, por el hilo asociado a un *weaving*, por el hilo del componente para procesar una petición directa, etc.), y puede darse el caso que dos hilos intenten encolar a la vez. Los objetos que se almacenan en dicha cola son instancias de la clase *ClassQueueAspect*, que almacena la información necesaria para ejecutar cada servicio cuando le llegue su turno:

- **itsDelegate**: contiene un delegado que apunta al servicio que debe ejecutarse.
- **parameters**: parámetros que se requiere el servicio a ejecutar.
- **callThread**: referencia al hilo que ha originado la petición de servicio, para el caso en que la petición sea síncrona (por lo que el hilo del cliente habrá quedado suspendido en espera de la finalización del servicio) y por tanto necesite ser despertado.

- **result**: instancia de la clase *AsyncResult*, en la que se almacenarán los resultados de la ejecución del servicio, para que puedan ser convenientemente procesados por una ejecución asíncrona.
- **attemptsToExecute**: es un contador que indica las veces que se ha intentado poner un servicio en ejecución sin éxito.
- **IncrementAttemptsToExecute()**: este servicio incrementa el atributo anterior .

La política de planificación de servicios, cuya implementación se encuentra en el servicio *AspectBase.DequeueMethod()*, es compleja, pues tiene que preservar el orden de las peticiones de servicio recibidas, teniendo en cuenta que existen servicios con mayor prioridad que otros (tal como se define en los subprocesos del aspecto), y que hay servicios que tal vez no se puedan ejecutar a causa de que el estado del protocolo del aspecto no es válido para su ejecución. El proceso seguido ha sido ejecutar los servicios en el mismo orden en que han sido encolados, y si a causa del protocolo no se pueden ejecutar, se incrementa el contador *attemptsToExecute* descrito anteriormente y se pasa a ejecutar el siguiente servicio. Cuando se consigue ejecutar con éxito un servicio, la siguiente iteración (en la que tal vez se hayan añadido nuevos servicios al final de la cola) volverá a empezar la ejecución del servicio desde el principio, dando la oportunidad de ejecutar los servicios que pudieran haber fallado anteriormente, ya que el protocolo del aspecto puede haber cambiado. Sin embargo, si el atributo *attemptsToExecute* es mayor que una constante definida en *AspectBase.maximumAttemptsToExecuteService*, el servicio finalmente es descartado lanzándose una excepción que, a través del elemento *AsyncResult* es transmitida al cliente. Todo este proceso está debidamente protegido frente al acceso concurrente mediante mecanismos de sincronización y ejecuciones atómicas, ya que mientras se está procesando un servicio, la cola del aspecto puede ser modificada con nuevos servicios a procesar.

La estructura de un aspecto se divide en una parte pública y una parte privada. La parte privada contiene las acciones a realizar por el servicio, cuya estructura es la comentada en el apartado 4.2.3.1. Los servicios privados se nombran igual que en la especificación del aspecto añadiendo como prefijo el símbolo "_" para distinguirlos de los métodos públicos. De esta forma, el servicio privado *Withdrawal* tendría la siguiente signatura:

```
private AsyncResult _Withdrawal(decimal quantity, ref decimal money) { ... }
```

La parte pública, que es por donde llegan las peticiones de servicio, se compone de los mismos servicios que publica el aspecto, con idéntico nombre y signatura, cuya finalidad es encolar las peticiones recibidas en la cola interna mediante una llamada al método heredado *EnqueueService()*. Siguiendo con el ejemplo anterior, la definición del método público para el servicio *Withdrawal* se realizaría de la siguiente manera:

```
public AsyncResult Withdrawal(decimal quantity, ref decimal money) {
   return EnqueueService(withdrawalDelegate, quantity, money);
}
```

El método heredado *AspectBase.EnqueueService* es el encargado de añadir la petición de servicio en la cola. Para ello, necesita como parámetros un delegado del servicio privado a ejecutar y una lista variable de los argumentos requeridos, devolviendo un objeto del tipo *AsyncResult*. Internamente se encarga de crear una instancia *ClassQueueAspect*, inicializarla convenientemente y encolarla. De esta forma, el código de la parte pública, que deberá generarse automáticamente por cada servicio definido por un aspecto, será mínimo.

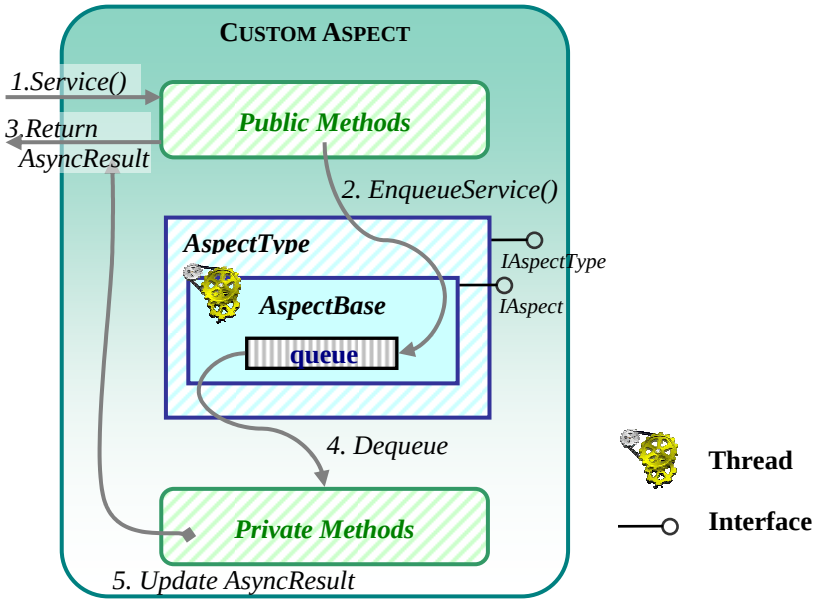

Figura 29 - Esquema de ejecución de un Aspecto

La creación de delegados en *C#* debe seguir tres pasos. En primer lugar deben declararse como tipos (con la misma signatura que el servicio al cual apuntan), en segundo lugar definir sus instancias, y en tercer lugar instanciarse con el método y el objeto donde se deben ejecutar:

```
// Definición del tipo de delegado WithdrawalDelegate
public delegate AsyncResult WithdrawalDelegate(decimal quantity,
     ref decimal money);
// Definición de una instancia del delegado
[NonSerialized] WithdrawalDelegate withdrawalDelegate;
// Instanciación del delegado anterior
withdrawalDelegate = new WithdrawalDelegate(this._Withdrawal);
```

Es por ello que también deberá generarse el código correspondiente a la definición de estos delegados por el compilador final. Además, como un delegado va siempre asociado a un servicio definido en una interfaz, se ha determinado que el lugar más apropiado para su generación es junto a la definición de las interfaces.

Finalmente, los servicios *startAspect*, *stopAspect* y *abortAspect* son necesarios para iniciar, parar o detener el hilo que procesa los servicios del

aspecto. Por ejemplo, cuando un componente va a ser movido junto con sus correspondientes aspectos, debe detenerse previamente la ejecución del aspecto. Además, debido a que en la cola de los aspectos se almacena el hilo del cliente en espera, y como los hilos no pueden ser serializados (y por tanto no pueden ser movidos), la movilidad de los aspectos queda limitada a que la cola de servicios a procesar esté vacía. Es por esta razón que el servicio *StopAspect* para detener el aspecto pasa por tres fases. En primer lugar no admite nuevas peticiones, por lo que éstas quedan almacenadas en el componente que las ha generado. En segundo lugar, termina de procesar las peticiones pendientes en la cola. En tercer lugar, el hilo se finaliza cuando todas las peticiones han sido procesadas.

Las clases que implementan los aspectos deben ser etiquetadas con el atributo *Serializable*, para permitir que su estado pueda distribuirse a otras máquinas mediante .NET Remoting. Sin embargo, debido a que en la versión actual de .NET los delegados no se comportan correctamente al ser serializados, éstos deben ser marcados como *NotSerialized* y vueltos a crear al llegar al destino.

Toda la funcionalidad asociada a un aspecto se empaqueta en un ensamblado y se almacena en la librería de tipos PRISMA, con el objetivo de facilitar la reutilización, la distribución por la red y la futura integración con un componente.

A continuación se muestra la estructura global de un aspecto en C#, omitiendo aquellas secciones de código no relevantes:

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Middleware;
using PRISMA.Attachments;
using PRISMA.Components.Ports;

namespace CuentaBancaria {
   // Definición del Aspecto Funcional "BankInteraction"
   [Serializable]
   Public class BankInteraction: FunctionalAspect, ICreditCardTransactions {

      Definicion de atributos PRISMA

      Definicion de los estados posibles del aspecto

      #region Definición de delegados
      [NonSerialized] WithdrawalDelegate withdrawalDelegate;
      // ...
      #endregion

      // Constructor del Aspecto: Inicialización
      public BankInteraction(decimal accountId) : base("BankInteraction") {

         // Estado inicial
         estado = protocolStates.BANKINTERACTION;

         // Inicialización de las vbles del aspecto
         this.numberId = accountId;
```

```
        // ...

        // Creación de la estructura de subprocesos para este aspecto
        SubProcessClass subProcessICredit = new SubProcessClass("SERVER");
        // ...

        // Asignacion del nuevo estado alcanzado
        estado = protocolStates.TRANSACTION;
      }

      #region Miembros Públicos de ICreditCardTransactions
      public AsyncResult Withdrawal(decimal quantity, ref decimal money) {
        return EnqueueService(withdrawalDelegate, quantity, money);
      }
      // ...
      #endregion

      #region Miembros Privados de ICreditCardTransactions
      private AsyncResult _Withdrawal(decimal quantity, ref decimal money) {
          ... }
      // ...
      #endregion

      // Sobreescribimos el metodo StartAspect para inicializar los delegados
      public override void StartAspect() {
        // Instanciacion de los delegados internos
        withdrawalDelegate = new WithdrawalDelegate(this._Withdrawal);
        // ...
        // Llamamos al padre
        base.StartAspect ();
      }
    }
}
```

4.2.4 Componentes y Conectores

Un componente PRISMA integra en su interior sus aspectos y ofrece una interfaz externa que es la agregación de las interfaces de sus aspectos. Esta interfaz es publicada al resto de elementos arquitectónicos a través de los puertos que forman el componente, que establecen qué interfaces serán publicadas. En .NET el componente se ha traducido por una clase que además de integrar dinámicamente en su interior los aspectos y los puertos de comunicación, también realiza dinámicamente los *weavings* entre los aspectos y redirecciona adecuadamente las peticiones de servicio (que pueden ser de entrada o de salida) entre puertos y aspectos. El funcionamiento de los conectores se ha implementado de la misma forma que los componentes.

El modelo de ejecución de los componentes proporciona los mecanismos necesarios para la gestión de la concurrencia, la gestión de aspectos, la gestión de weavings y la redirección de servicios desde y hacia los puertos. En los siguientes apartados se describe cómo se han implementado los distintos procesos. Finalmente, se muestra la correspondencia entre los componentes y conectores PRISMA y su implementación en C#.

4.2.4.1 Modelo de ejecución

De la misma forma que en los aspectos se ha definido una interfaz común con los servicios básicos que deben ofrecer, para los componentes se ha definido la interfaz *IComponent* (ver Figura 30), cuyos atributos y servicios se describen a continuación:

- **componentName**, **componentThread** y **middlewareServer**: Atributos que almacenan el nombre, una referencia al hilo del componente y una referencia al middleware, respectivamente.
- **isStopping, isMoving**: Atributos que permiten comprobar si el componente está en proceso de detenerse o de moverse, por lo que no aceptará nuevas peticiones.
- **inPorts, outPorts**: Atributos que almacenan las referencias a los puertos de entrada (recepción de peticiones de servicio) y a los puertos de salida (peticiones de servicio a otros componentes externos) que integra el componente.
- **Start, Stop, Abort**: Servicios que permiten iniciar, parar o detener al componente.
- **EnqueueService**: Método invocado por los puertos para añadir las peticiones de servicio entrantes en la cola del componente.

- **ProcessService**: Método privado, invocado por el hilo de ejecución del componente para procesar una petición de servicio añadida a la cola.
- **AddAspect, RemoveAspect**: Servicios para añadir o eliminar un aspecto.
- **GetAspect**: dado el tipo de un aspecto, este servicio comprueba si existe en el componente, y en caso afirmativo, devuelve su referencia.
- **AddWeaving**, **RemoveWeaving**: Servicios para añadir o eliminar un weaving entre dos aspectos, respectivamente.
- **IsWeaved**: Servicio de consulta que permite comprobar si un aspecto tiene un *weaving* con otro aspecto. Es necesario consultar previamente si un aspecto tiene weavings relacionados antes de eliminarlo.

Todos estos servicios son implementados por la clase *ComponentBase*, de la cual todos los componentes y conectores heredan su comportamiento.

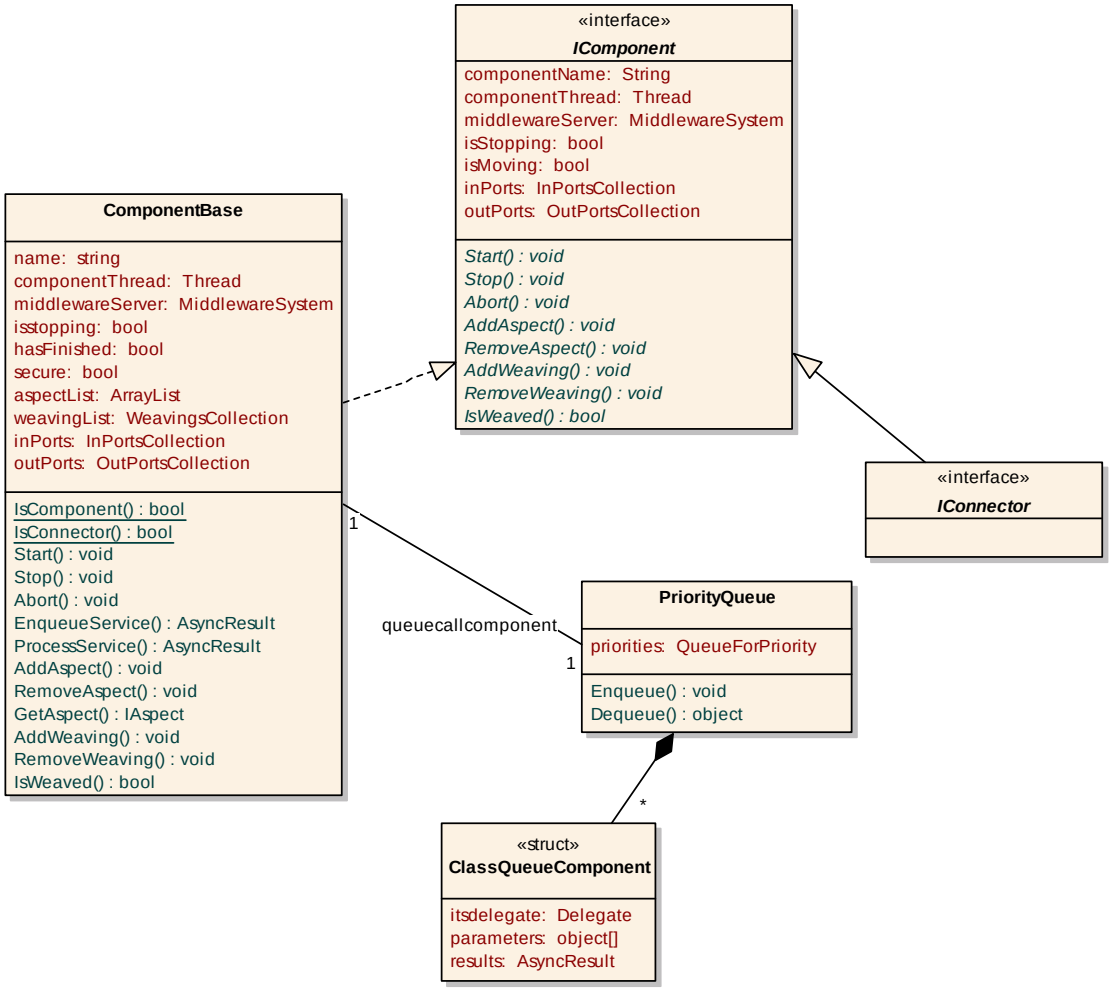

Figura 30 - Interfaces IComponent, IConector y clase ComponentBase

Los conectores heredan de *ComponentBase* debido a que su comportamiento es idéntico al de los componentes, salvo que en lugar de puertos tienen roles y en lugar de un aspecto funcional tienen uno de coordinación. Como no se han detectado atributos y servicios básicos adicionales a los de un componente, el conector se ha definido de la misma forma que un componente con la salvedad de que implementa la interfaz *IConnector*. Esta interfaz a su vez hereda de *IComponent* y tan sólo se utiliza como mecanismo de identificación frente a componentes. Por ello, se han añadido dos métodos de clase en *ComponentBase* que permiten comprobar cuándo una entidad que implementa *IComponent* es componente o conector. Estos servicios son *IsComponent()* y *IsConnector()*.

Las diferencias de comportamiento entre componentes y conectores se han resuelto de diferentes maneras. En la implementación realizada, los roles PRISMA se traducen a puertos, cuya implementación se verá más adelante, relegándose al *middleware* la tarea de comprobar que los conectores se comuniquen sólo con componentes (y viceversa). Por otra parte, la comprobación de que un conector deba tener un aspecto de coordinación (y un componente el aspecto funcional) se realiza en el momento de añadir un nuevo aspecto al componente o conector mediante la llamada al servicio *AddAspect*. Este método, antes de agregar el aspecto al componente comprueba que se cumplen las precondiciones establecidas en el tipo del aspecto, ejecutando el servicio *IAspectType.CheckPreconditions*, como se comentó en el apartado 4.2.3.2.

Un componente está formado por los puertos de entrada y de salida, por una cola de prioridades en la que se almacenan los servicios a procesar, un gestor de *weavings* y los aspectos (ver Figura 31).

- Los puertos de entrada (*inPorts*) son entidades que publican a los demás elementos arquitectónicos los servicios que ofrece el componente. Publican la interfaz que implementa un aspecto integrado en el componente y encolan las peticiones recibidas en la cola interna.
- Los puertos de salida (*outPorts*) son aquellos elementos que proporcionan los mecanismos necesarios para enviar peticiones de servicio a otros componentes. Cada puerto de salida únicamente admite servicios de una determinada interfaz y sólo se conectará con aquellos componentes cuyos puertos de entrada publiquen idéntica interfaz. Tanto los puertos de entrada como los de salida son seguros para el acceso multihilos.
- La cola de prioridades (*PriorityQueue*) almacena las diferentes peticiones de servicio recibidas hasta que puedan ser redirigidas por el componente hacia el aspecto correspondiente. En caso de conflicto, primero se entregará al aspecto el servicio más prioritario.
- El gestor de *weavings* (*WeavingsCollection*) se encarga de comprobar, por cada servicio solicitado a un aspecto, si tiene weavings o no. En

caso negativo, se redirige la petición al aspecto correspondiente. En caso afirmativo, se procesa el weaving de forma síncrona.

- Los aspectos se almacenan en una lista dinámica del componente, existiendo únicamente uno de cada tipo.

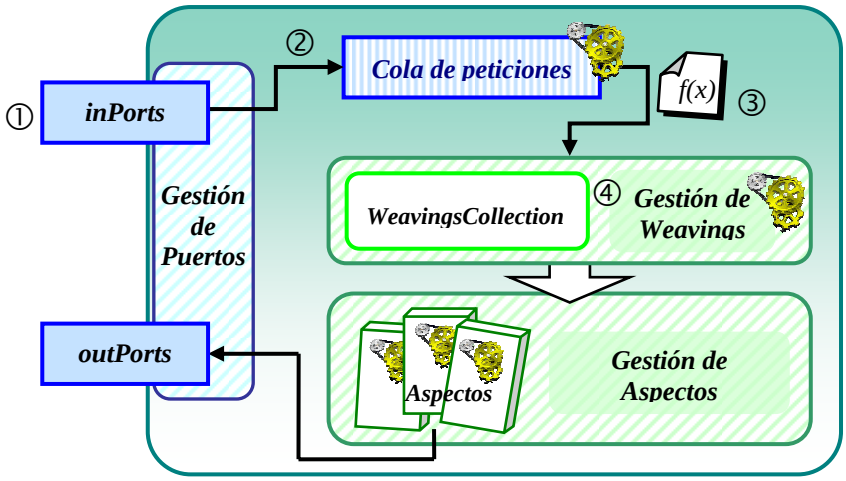

Figura 31 - Estructura interna de un componente en .Net

Cuando un componente externo solicita un servicio al componente, éste es solicitado al puerto que publica la interfaz de dicho servicio (paso 1, Figura 31), que se encarga de reenviarlo a la cola interna del componente (paso 2, Figura 31). Ésta es una cola ordenada por prioridades (clase *PriorityQueue*) en la que se almacenan los servicios a ejecutar por los aspectos. Dichas prioridades corresponden a las declaradas por los subprocesos de los aspectos, mediante los cuales puede establecerse una ordenación de los servicios a ejecutar. La información requerida por cada servicio a ejecutar se encapsula en objetos de tipo *ClassQueueComponent*. Estos objetos contienen un delegado hacia el método del aspecto a ejecutar, un vector con los parámetros necesarios y una referencia al objeto *AsyncResult* creado para devolver el resultado cuando el servicio sea procesado.

Debido a que a un componente le pueden llegar múltiples peticiones de servicio por los distintos puertos que publica, y todos confluyen en la cola interna, ésta se ha protegido para el acceso concurrente mediante el uso de monitores, de tal forma que únicamente un hilo pueda estar a la vez añadiendo servicios en la cola. El reenvío de las peticiones de servicio almacenadas a los aspectos correspondientes, y en su caso, la aplicación de los diferentes *weavings*, es realizada por un hilo de ejecución propio del componente (paso 3, Figura 31). Cuando un servicio va a ser procesado, en primer lugar se comprueba si tiene weaving asociado. Si no lo tiene la llamada es directa, ejecutándose el delegado de forma asíncrona, con la finalidad de que el hilo del componente pueda procesar las siguientes peticiones los más pronto posible. Si lo tiene, el servicio es redirigido al gestor de weavings (paso 4, Figura 31), el cual se encargará de procesar todos los servicios relacionados, liberando al hilo del componente de esta tarea.

De la misma forma que en los aspectos, el hilo de ejecución del componente es puesto en ejecución cuando el método *Start* es invocado por el *middleware*, y es detenido cuando se invocan los servicios *Stop* (parada segura) o *Abort* (detención completa). La parada o detención de un componente implica la parada o detención de cada uno de los aspectos que lo forman, invocando secuencialmente los servicios *StopAspect()* o *AbortAspect*(), respectivamente.

La parada segura (*Stop*) de un componente es posible mediante una serie de flags internos que permiten sincronizar la parada de las distintas operaciones hasta alcanzar un estado seguro que permita posteriormente reanudar la ejecución en el punto que se había quedado. Un componente está en un estado seguro cuando no quedan peticiones en las colas internas de sus aspectos (pues dichas peticiones no son serializables, como se comentó), no existe ningún servicio en ejecución en los aspectos, ni tampoco se interrumpe ninguna inserción de servicios en la cola. El primero de ellos es el flag *secure*, el cual se utiliza para impedir que el componente sea parado o movido mientras está realizando operaciones atómicas que no pueden ser interrumpidas, como encolar un servicio. El flag *isStopping* indica que el componente va a detenerse y es activado cuando se invoca el servicio *Stop()*. El flag *isMoving* indica que el componente va a moverse y es activado por el aspecto de distribución cuando éste lo estime oportuno.

Cuando uno de estos dos flags está activo, se restringe el encolado de nuevas peticiones de servicio y únicamente son permitidas aquellas provenientes de los aspectos internos. Las peticiones de servicio rechazadas son notificadas al cliente mediante una excepción del tipo *IsMovingComponentException*, que normalmente se traducirá en un reintento por parte de los *attachments* hasta que el componente esté de nuevo activo. En cambio, si la petición es realizada por un aspecto interno, ésta es aceptada y encolada con la máxima prioridad, pues los servicios pendientes de ejecución de un aspecto deben finalizar para que el componente pueda moverse. El hilo de ejecución del componente también deja de reenviar peticiones a los aspectos, excepto los de máxima prioridad. Ésto es necesario para evitar interbloqueos, ya que cuando el componente va a pararse, ha de esperar a que los aspectos acaben de procesar los servicios pendientes. Sin embargo, si un servicio en ejecución de un aspecto requiere otro servicio del componente, y éste ya no admite más peticiones, el aspecto se quedaría a la espera indefinidamente, con lo componente y aspecto quedarían en espera mutua sin llegar nunca a poder detenerse en un estado consistente.

4.2.4.2 Gestión de Aspectos

Para unir los aspectos al componente se consideraron dos aproximaciones:

- **Estática**
  La aproximación más sencilla y eficiente consiste en instanciar cada aspecto en una variable del tipo del aspecto (*FunctionalAspect, DistributionAspect¸* etc.) en el momento de la creación del componente, de forma totalmente estática. Esto implica que los aspectos deberían crearse en el código generado a partir de las especificaciones PRISMA, sin utilizar la herencia para heredar el comportamiento. Por lo tanto, para añadir nuevos aspectos o eliminarlos, se debería emitir código o recompilar el componente. El inconveniente de esta aproximación es que proporciona muy poca flexibilidad al cambio.
- **Dinámica:**
  supone la utilización de una lista dinámica (un *ArrayList*) y una serie de métodos para añadir o eliminar aspectos de dicha lista, cumpliendo la restricción de que no existan dos aspectos del mismo tipo instanciados en un componente. La invocación de servicios de un aspecto supone recorrer la lista dinámica para obtener la referencia a un aspecto. El problema de esta solución es el incremento del coste computacional que se produce cuando se realizan muchas llamadas a los aspectos. La principal ventaja es la gran flexibilidad que aporta para añadir/eliminar aspectos.

De las dos aproximaciones anteriores se ha implementado una solución intermedia, con la eficiencia de tener accesible en una variable la referencia hacia un aspecto y la flexibilidad proporcionada por las listas dinámicas. Como en PRISMA el número de tipos de aspectos no es limitado, éstos se han almacenado en una lista dinámica, de forma que puedan ser accedidos todos los aspectos de forma secuencial para realizar diferentes operaciones, como la detención de los aspectos, la comprobación de que un tipo no ha sido ya agregado, etc. Por otra parte, como los puertos de entrada publican únicamente una interfaz implementada por un aspecto, en cada puerto se han definido una serie de delegados que apuntan a cada uno de los servicios ofrecidos por el aspecto asociado. De esta forma, cuando se crea un *inPort* se instancian los delegados que apuntan a cada uno de los respectivos servicios de los aspectos, con lo que la invocación de un aspecto es directa y no necesita recorrer cada vez la lista dinámica de aspectos.

Los aspectos pueden ser agregados o eliminados en tiempo de ejecución sin la necesidad de detener el componente. La agregación de aspectos se realiza mediante la llamada al método *AddAspect(IAspect ref_aspect).* Este método en primer lugar comprueba que el aspecto que va a agregar no sea de un tipo de aspecto de los que ya han sido agregados al componente. En segundo lugar, comprueba que el componente cumple las precondiciones especificadas en el tipo del aspecto. Después actualiza las referencias del

aspecto hacia el componente y el *middleware* y lo añade en la lista dinámica de aspectos. Finalmente, aunque no está implementado por no encontrarse entre los objetivos de este proyecto, se emitiría el código correspondiente en el constructor del componente para garantizar la persistencia de los cambios. Sin embargo, los aspectos no tendrán visibilidad hacia el exterior del componente hasta que no se creen los puertos relacionados que publiquen su interfaz. Para añadir el aspecto *BankInteraction* al componente *Account* se invocaría al método *AddAspect* pasándole como parámetro una instancia del tipo *IAspect*, como el aspecto que se definió anteriormente:

```
// Creación del aspecto funcional
AddAspect(new BankInteraction(numberID, customer, address, money));
```

La eliminación de aspectos se realiza mediante la llamada al método *RemoveAspect(Type AspectType, bool removeWeavings)*, que permite especificar el tipo de aspecto a eliminar y si se deben o no eliminar los weavings asociados. En primer lugar se eliminan los aspectos de la lista dinámica y los puertos correspondientes que estaban asociados al aspecto, con la finalidad de que no se puedan recibir nuevas peticiones. En segundo lugar, se realiza una detención segura del aspecto, dando un margen de actividad para finalizar los servicios que estuviera ejecutando. Si ese margen es superado, se obliga al aspecto a detenerse por completo. Finalmente, se debería emitir código para garantizar la persistencia de los cambios en el disco.

El componente también proporciona el método *GetAspect(Type AspectType)* que permite obtener la referencia del aspecto de un tipo concreto o comprobar si dicho tipo de aspecto ha sido agregado al componente. Por ejemplo:

```
// Obtenemos referencias a los aspectos
IAspect functionalAspect = GetAspect(typeof(FunctionalAspect));
```

4.2.4.3 Weavings

Como se ha comentado anteriormente, los *weavings* son mecanismos que permiten que la ejecución de un servicio de un aspecto concreto se asocie antes, después o en lugar de un servicio ofrecido por otro aspecto. Una diferencia importante respecto a otras tecnologías orientadas a aspectos es que esta funcionalidad es especificada en los componentes y no en los aspectos, de forma que los aspectos desconocen a qué otro aspecto pueden ser enlazados, con el objetivo de facilitar la reutilización de aspectos.

♦ Estrategias de implementación

No existen trabajos previos de implementación en .NET para dar soporte a los weavings a nivel de código fuente. Las aproximaciones estudiadas en el Capítulo 2 que trabajaban a este nivel únicamente realizaban un *weaving* estático, como se describe a continuación. En cambio, ninguna ha proporcionado un diseño capaz de soportar el *weaving* dinámico.

Por esta razón, para la implementación de los weavings en .NET se han diseñado cuatro estrategias distintas:

- *Inserción estática de código*

  Los weavings pueden traducirse directamente en llamadas a los servicios de los aspectos requeridos, pero eso supone que cuando se desee insertar o eliminar un weaving no pueda modificarse fácilmente el código insertado, ya que está mezclado con el código dedicado a otras tareas. La ventaja de esta estrategia es el incremento del rendimiento del sistema en ejecución, frente a otras alternativas, ya que son simplemente llamadas a métodos.

  No obstante, esta estrategia podría llevarse a cabo asociando a cada método un atributo que ofreciese las funciones *after, before* e *instead* y que contuviesen el código necesario para llamar a los métodos de los aspectos correspondientes. Con esto se conseguiría que los weavings de dicho método estuviesen aislados en el código del atributo. Sin embargo, a pesar de esta ventaja, su modificación seguiría haciéndose emitiendo código para el cuerpo de dichas funciones.

- *Intercepción de código mediante Proxies*

  .NET permite mediante el uso de *ContextBoundObjects* y *ContextAttributes* utilizar mecanismos para separar la ejecución de una clase determinada de las demás clases, mediante la creación de un nuevo contexto de ejecución. Este contexto permite el filtrado de las peticiones de servicio dirigidas a dichas clases aisladas, es decir, el preprocesado y postprocesado de código previo a la ejecución de los servicios ofrecidos por dichas clases. De esta manera, cada aspecto debe heredar de *ContextBoundObject* y etiquetarse con un *ContextAttribute*, y el código del componente sería el encargado de rellenar adecuadamente dichas estructuras con los weavings a realizar. Esto se implementaría situando una llamada al aspecto destino en el preproceso (para el caso de weavings *before*) o en el postproceso (para el caso de weavings *after*), consiguiendo que cada servicio ofrecido por el aspecto sea interceptado y procesado adecuadamente. No obstante, esta aproximación supone interceptar todos los aspectos de un componente y si hay varios componentes, esto supone que cada clase se ejecute en un contexto diferente, con la sobrecarga que esto conlleva para el sistema.

- *Uso de eventos definidos en el código del aspecto*

  Otra aproximación consiste en definir tres eventos asociados a cada servicio del aspecto: *before*, *after* o *instead*. En el código del servicio se comprueba si se ha suscrito algún delegado a dichos eventos, y en caso afirmativo, se ejecuta el correspondiente delegado. La tarea de añadir/eliminar weavings es realizada por el componente, suscribiendo eventos a dichos métodos mediante el uso de delegados que apuntan al servicio a ejecutar del siguiente aspecto. El principal inconveniente es que se ha de modificar el código de los aspectos para publicar los eventos, pero permite que ante cualquier petición de servicio externa se le pase el control al componente para realizar el procesado necesario.

- *Uso de una lista de delegados*

  Esta aproximación supone que todas las peticiones de servicio se realicen a través del componente, y por tanto, el procesado de los weavings se hace previamente a la llamada al aspecto correspondiente. La idea general es utilizar una serie de listas dinámicas anidadas para finalmente almacenar los delegados asociados a cada tipo de weaving. Esta aproximación proporciona la suficiente flexibilidad para añadir/eliminar dinámicamente los weavings (sin emitir código), pero presenta la limitación de sólo aplicarse a nivel de componente; es decir, si en un aspecto se dispara un *trigger*, al no pasar por el componente, no se podría interceptar su llamada para aplicarle un weaving (a no ser que se modificase el código del aspecto, provocando que la ejecución de un *trigger* necesariamente deba redirigirse al componente). Por otra parte, proporciona una estructura para el almacenamiento conjunto de todos los weavings sin entremezclarse con el código del componente.

De entre las estrategias vistas, la inserción estática de código es la que presenta mejor rendimiento, y la intercepción de código la más flexible. Con el objetivo de encontrar un equilibrio entre flexibilidad y rendimiento, se ha implementado la aproximación que utiliza listas de delegados para la gestión de weavings (*WeavingsCollection*). El principal inconveniente de la implementación realizada es que la estructura para almacenar los diferentes delegados es compleja, formada una lista enlazada con tres niveles de profundidad, lo que provoca que para comprobar si un servicio tiene weaving se tengan que realizar hasta tres búsquedas por las diferentes listas. Esto se puede optimizar implementando algoritmos de búsqueda binaria o de búsqueda rápida.

- **La estructura de almacenamiento de los weavings**

En la definición de un weaving existen tres elementos clave por los cuales se puede clasificar la información de forma jerárquica, y con ello optimizar las búsquedas. En primer lugar, un weaving se asocia a un **tipo de aspecto**, y en concreto a un servicio de dicho aspecto. A este servicio se le denominará **método original** pues su invocación desencadenará la invocación de otros servicios asociados a través del weaving. El método original puede tener asociados hasta tres **tipos de weavings** distintos: *after*, *before* e *instead*. Se incluyen los tipos de weavings condicionales *afterIf* y *beforeIf* como un subtipo del weaving *after* y *before*, respectivamente. Al servicio a ejecutar como consecuencia de la aplicación de un weaving se le denominará **método destino**, pues puede reemplazar al método original (*instead*), modificar los parámetros de entrada (*before*) o modificar los parámetros de salida (*after*). Tanto el método original como el método destino se almacenarán en forma de delegados. Por tanto, y de forma general, navegar por la lista dinámica supone atravesar tres niveles hasta llegar finalmente al delegado que debe ejecutarse (método destino): el tipo de aspecto, el método original y el tipo de weaving. A continuación se describirá cómo se ha implementado esta estructura.

WeavingsCollection (ver Figura 32) es la clase que, además de proporcionar el comportamiento del gestor de weavings, contiene la lista dinámica descrita (*weavingList*) junto a las funciones necesarias para su manejo. *weavingList* almacena los distintos tipos de aspectos que tienen weaving asociado y representa al primero de los niveles descrito anteriormente. Cada nodo de esta lista es una instancia de la clase *AspectTypeNode*, que contiene el tipo de aspecto que representa (*aspectType*) y otra lista dinámica (*weavingAspectList*) con los distintos servicios del aspecto: esta lista representa el segundo nivel. Los elementos de esta segunda lista son instancias de *WeavingNode* y como información identificativa guardan el nombre del servicio que representan (el método original), un delegado a dicho servicio para posteriormente poderlo invocar dinámicamente, y tres listas dinámicas adicionales correspondientes a los weavings *after, before* e *instead*. Estas listas representan el tercer nivel de profundidad. En ellas se almacenan instancias de *WeavingMethod*, cuyos atributos y métodos son los siguientes:

- **methodDelegate**: Atributo que contiene el delegado que apunta hacia el método destino, es decir, el servicio cuya invocación será provocada por el weaving.
- **origMethod**: Atributo que contiene el delegado del método origen, es decir, el servicio que ha provocado la activación de los weavings.
- **weavingType**: Atributo que define el tipo de weaving que representa el nodo, y para el caso de los weavings condicionales, define la condición que debe cumplirse para aplicarse el weaving. Se describirá más adelante.

- **functions**: Atributo que contiene una instancia de *DelegateFunctionsCollection*, y define las funciones de transformación que deberán aplicarse a cada parámetro antes de ejecutar el método destino.
- **inputMethodParameters y outputMethodParameters**: Atributos que definen los tipos de parámetros que requiere el método destino para ejecutarse. Se describirá más adelante cuando se describan las funciones de transformación, pues son las que justifican la existencia de estos atributos.

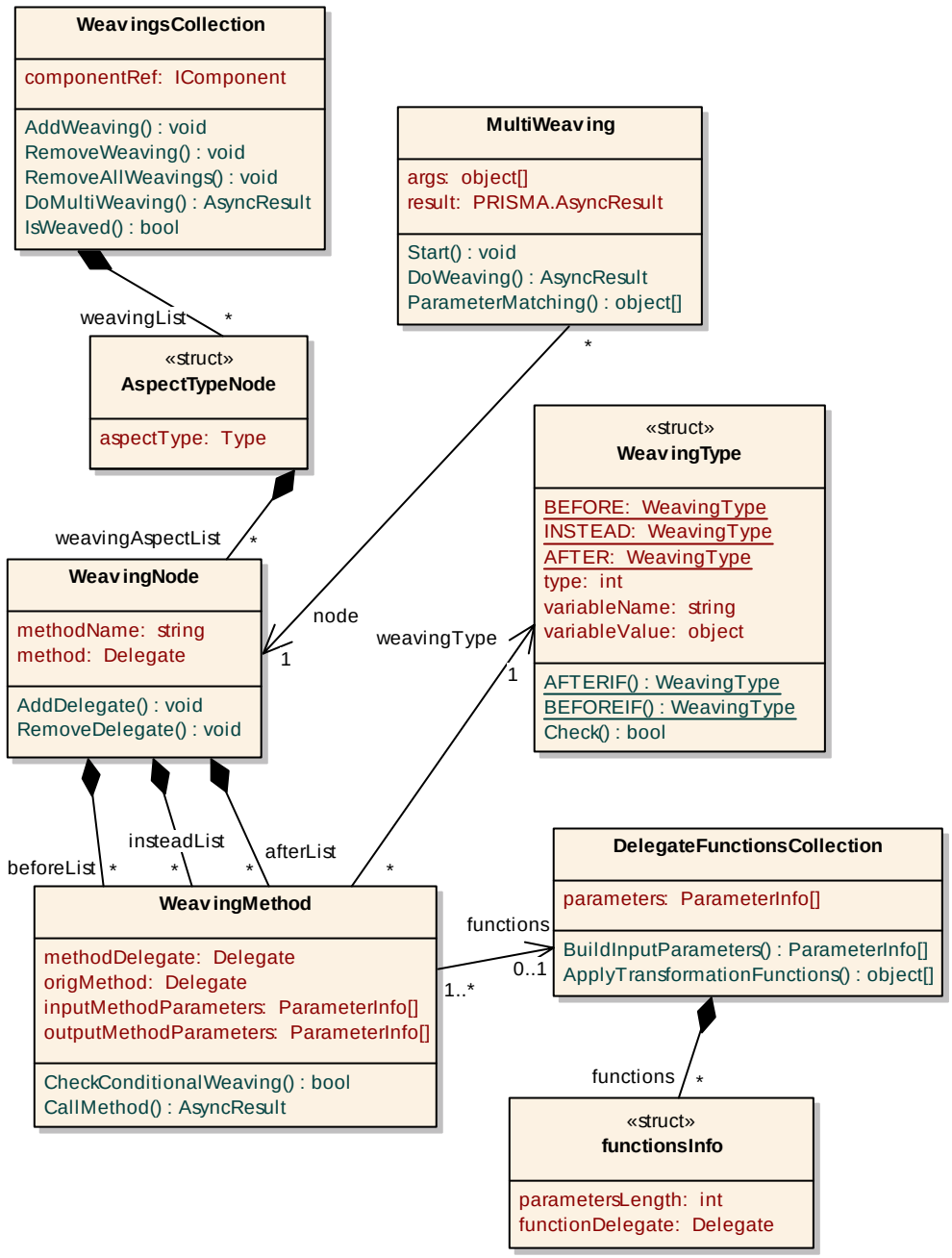

Figura 32 - Clases asociadas al Weaving

- **CheckConditionalWeaving(args: object[])**: Servicio que comprueba si se cumple la condición especificada por los weavings condicionales. Como parámetros requiere la lista de argumentos del método origen.
- **CallMethod(args: object[])**: Servicio que ejecuta el método destino, aplicándole previamente las funciones de transformación definidas. Como parámetros requiere la lista de argumentos del método origen.

La clase *WeavingType* permite representar el tipo de weaving especificado. Se compone de una serie de atributos de clase que contienen instancias de sí misma pero inicializadas adecuadamente para representar diferentes tipos de weaving: *after, afterIf, before, beforeIf* o *instead*. Internamente, un atributo numérico (*type*) permite diferenciarlos entre sí. Los weavings condicionales requieren como parámetros adicionales el nombre de la variable cuyo valor se ha de comprobar (*variableName*) y el valor que ha de tener dicha variable para que la condición se evalúe a cierta (*variableValue*). El método *Check(object variable)* permite comprobar que el valor de la variable pasada como parámetro es el que evalúa a cierta la condición. Sin embargo, ha quedado como tarea futura el implementar los demás operadores de comparación (‘<’, ‘>’, ‘≤’, ‘≥’, ‘≠’) en los weavings condicionales para aumentar la expresividad del modelo. Para ello, basta con añadir un parámetro adicional en la especificación de los weavings condicionales que defina el tipo de comparación a realizar.

♦ Creación y Eliminación de weavings

Una vez vistos los distintos elementos que forman un weaving, faltaría describir cómo los métodos *AddWeaving* y *RemoveWeaving* permiten definir, de forma intuitiva, la creación y eliminación de weavings. El método *AddWeaving* permite añadir nuevos weavings a la lista dinámica, creando un nuevo nodo *WeavingMethod* e insertándolo en la posición correspondiente dentro de la jerarquía. La signatura del método es la siguiente:

```
public void AddWeaving(IAspect origAspectInstance, string origMethod,
     WeavingType weavingType, IAspect destAspectInstance,
     string destMethod, DelegateFunctionsCollection functions)
```

El parámetro *origAspectInstance* es una referencia a un aspecto ya creado que contiene el método *origMethod*, al cual se le va aplicar un weaving de tipo *weavingType*. El servicio que se ejecutará (*destMethod*), según determine el weaving, pertenece al aspecto *destAspectInstance*. El parámetro *functions* permite especificar, de forma opcional, las funciones que se aplicarán para transformar los parámetros del método original a los requeridos por el método destino.

Sin embargo, como se puede observar, para definir un nuevo weaving no se proporciona el tipo del aspecto, sino un aspecto ya agregado al componente (y por tanto una instancia de la clase del aspecto). Esto es necesario debido a

que los delegados únicamente se pueden crear para instancias de clases, y por tanto para crear dinámicamente el delegado correspondiente al método *origMethod* se requiere la instancia de la clase del aspecto al cual pertenece el método. Esto presenta como limitación que los weavings no puedan existir si el aspecto correspondiente es eliminado. De esta forma, si un aspecto se elimina del componente y es sustituido por otro del mismo tipo con los mismos servicios, los weavings deberían volverse a crear. Esto queda como una tarea futura a resolver.

El método *RemoveWeaving* permite eliminar el weaving asociado a un método concreto. Los parámetros requeridos son el aspecto (*aspect*) al cual pertenece el método *origMethod* cuyo weaving se va a eliminar. El parámetro *weavingType* permite indicar qué tipo de weaving se va a eliminar y *destMethod* el método asociado a dicho weaving.

```
public void RemoveWeaving(IAspect aspect, string origMethod,
     WeavingType weavingType, string destMethod)
```

A continuación se muestra como un weaving PRISMA se traduce a un weaving de la implementación. Por ejemplo, el siguiente weaving en PRISMA especifica que al retirar dinero de la cuenta (*Withdrawal*) se actualice adecuadamente el formulario asociado (aspecto de presentación, *ClientInteraction*) con el nuevo saldo restante:

```
Functional Aspect import BankInteraction;
  Weaving
    BankInteraction.Withdrawal(quantity: decimal,money: decimal);
      before ClientInteraction.Show_money(money: decimal);
  End_Weaving;
```

En la implementación, dicha especificación se correspondería con:

```
AddWeaving( functionalAspect, "Withdrawal", WeavingType.BEFORE,
            presentationAspect, "Show_money");
```

De forma similar, la especificación de un weaving condicional para especificar que si se alcanza el saldo cero en la cuenta el usuario sea alertado con un mensaje de advertencia correspondería con:

```
AddWeaving( functionalAspect, "Withdrawal",
              WeavingType.BEFOREIF("money",(decimal) 0),
            presentationAspect, "DisableWithdrawal");
```

Por otra parte, el método *IsWeaved* permite comprobar si el método de un aspecto tiene weaving asociado. Este método es invocado por el hilo del componente para comprobar si el servicio que va a ser redireccionado al aspecto tiene weaving asociado. La signatura del método es la siguiente:

```
public bool IsWeaved(IAspect origAspect, string origMethod)
```

- **El tratamiento de los parámetros**

Por otra parte, también ha sido necesario definir una política adecuada para el paso de argumentos. Cuando se invoca el servicio de un aspecto (método original), se le proporcionan los argumentos necesarios para su ejecución. Si éste tiene weaving asociado, sus argumentos también serán proporcionados a los servicios cuya invocación desencadene el weaving (métodos destino).

El problema reside en que los métodos destino pueden tener distinto número y tipo de argumentos que el método original invocado. Esto se ha solucionado pasando a cada método destino los parámetros del método original que concuerdan en nombre y tipo.

Sin embargo, esta estrategia introduce una limitación para la definición de weavings. Si se desea definir un weaving de forma que uno o más parámetros del método origen deban ser proporcionados a otro método destino, éste deberá definirse de forma que el nombre del parámetro que vaya a ser proporcionado por el método origen coincida en nombre y tipo. Por ejemplo, nótese cómo en la definición del siguiente weaving los parámetros "*newAdd*" de los servicios *ChangeAddress* y *Move* concuerdan en nombre y tipo, debido a que al servicio *Move* se le proporcionará el valor que se le proporcione al servicio *ChangeAddress*.

```
Distribution Aspect Import ExtMbile;
  Weaving
    BankInteraction.ChangeAddress(newAdd: string)
      before ExtMbile.Move(newAdd: string);
  End_Weaving;
```

Otra estrategia posible podría no considerar el nombre del parámetro y realizar la comparación únicamente por el tipo y la posición. Sin embargo, no ha sido implementada ya que puede producir efectos indeseados si el método origen tiene varios argumentos del mismo tipo y su posición no coincide con la que define el método destino.

Por otra parte, independientemente de la estrategia empleada, si el método destino tiene más argumentos que el método original surge el problema de qué valores proporcionar a estos argumentos adicionales. Para resolverlo se han creado las funciones de transformación de parámetros.

Las funciones de transformación de parámetros permiten asociar a cada parámetro de un método destino una función. Dicha función toma como entrada uno o varios parámetros definidos para el método origen y devuelve un objeto del mismo tipo que el parámetro del método destino al cual está asociada. Por ejemplo, siguiendo con el ejemplo anterior, si se quisiera realizar un weaving entre el método *ChangeAddress(newAddress: string*) del aspecto funcional y el método *Move(newLoc: LOC)* del aspecto distribución, podría definirse una función con la siguiente signatura:

```
LOC ChangeAddressToLOC(newAdd: string);
```

El funcionamiento de dicha función sería el de transformar una cadena de texto al tipo LOC. Por tanto, la definición del weaving sería la siguiente:

```
Distribution Aspect Import ExtMbile;
  Weaving
    BankInteraction.ChangeAddress(newAdd: string)
      before ExtMbile.Move(ChangeAddressToLOC(newAdd: string));
  End_Weaving;
```

Obsérvese cómo los parámetros del método origen (*ChangeAddress*) y de la función definida concuerdan en nombre y tipo, y que el valor de retorno de ésta coincide con el tipo del parámetro del método destino (*Move*).

Las funciones de transformación de parámetros se almacenan en una instancia de la clase *DelegateFunctionsCollection*, cuyos atributos y servicios son los siguientes (ver Figura 32):

- **parameters**: Atributo que almacena la información relativa a los parámetros del método destino, es decir, el método cuyos parámetros van a ser proporcionados por los resultados de las funciones definidas.
- **functions**: Vector de instancias *functionsInfo* del mismo tamaño que el número de parámetros del método destino. Define la función que se aplicará sobre cada parámetro.
- **DelegateFunctionsCollection**: Servicio constructor de la clase. Se le proporcionan como parámetros el aspecto que contiene el método destino y el nombre del método destino. Inicializa adecuadamente los atributos *parameters* y *functions*.
- **this[“nombre_parámetro”]**: Indexador que permite asociar una función de transformación al parámetro cuyo nombre se especifica.
- **BuildInputMethodParameters()**: Servicio que construye la lista de parámetros resultante de la agregación de los tipos requeridos por cada una de las funciones de transformación definidas. Es utilizado exclusivamente para inicializar la propiedad *inputMethodParameters* de la clase *WeavingMethod*.
- **ApplyTransformationFunctions(args: object[])**: Servicio que aplica las funciones de transformación a la lista de parámetros pasada como argumento, devolviendo la lista de parámetros que requiere el método destino.

La estructura *functionsInfo* representa a una función de transformación y únicamente contiene un delegado que apunta a la función a ejecutar (atributo *functionDelegate*) y el número de parámetros que requiere dicha función (atributo *parametersLength*). Siguiendo con el ejemplo anterior, el siguiente fragmento de código muestra cómo asociar la función de transformación *ChangeAddressToLoc* al parámetro *newLoc* del método *Move* del aspecto de distribución:

```
// Creamos la estructura para alojar func. de transform. asociadas a "Move"
DelegateFunctionsCollection functions =
```

```
        new DelegateFunctionsCollection(distributionAspect, "Move");

// Asociamos las funciones de transformación a cada parámetro.
// Para ello, debemos proporcionar un delegado hacia la función a aplicar
functions["newLoc"] = new ExtMbile.ChangeAddressToLOCDelegate(
                ((ExtMbile) distributionAspect).ChangeAddressToLOC);

// Creamos el Weaving, especificando que se aplican func. de transform.
AddWeaving(functionalAspect, "ChangeAddress", WeavingType.BEFORE,
                   distributionAspect, "Move", functions);
```

Una vez visto cómo se almacenan las funciones de transformación en la estructura de weavings, falta por describir cómo se aplican a los parámetros. Para ello, en primer lugar es necesario introducir el tipo de datos *ParameterInfo*. Éste se encuentra definido en el espacio de nombres de .NET System.Reflection, y proporciona información sobre los atributos de un parámetro, como el nombre, el tipo, si es de entrada o de salida, etc. La información de los parámetros de un método es almacenada en vectores *ParameterInfo* y se obtiene a través de la reflexión. Esta información es utilizada para poder realizar el filtrado de parámetros, como se describe a continuación.

Cuando el método destino va a ser invocado por el weaving, hay que proporcionarle la lista de parámetros necesaria para su ejecución. Ésta viene determinada por el método origen, que puede tener distinto número y tipo de argumentos, por lo que antes de invocar al delegado del método destino será necesario filtrar dicha lista de parámetros con los requeridos. Esta tarea es realizada por el método privado *ParameterMatching*, definido en la clase *MultiWeaving,* cuya signatura es la siguiente:

```
private object[] ParameterMatching(
            ParameterInfo[] origParameters, object[] origArgs,
            ParameterInfo[] targetParameters, object[] targetArgs);
```

Como parámetros de entrada toma la definición de los parámetros del método origen y del método destino (de tipo *ParameterInfo*), así como un vector con los argumentos del método origen (*origArgs*), que serán filtrados y copiados en el vector de argumentos del método destino (*targetArgs*), en el orden correcto que espera el método. Básicamente, esta función va comparando los parámetros del método origen en nombre y tipo con los del método destino, seleccionando únicamente los que necesita el método destino. Si algún parámetro requerido por el método destino no es encontrado, se deja el hueco correspondiente en el vector *targetArgs*, el cual será interpretado como el valor por defecto del tipo esperado (por ejemplo, si se esperaba un entero, el hueco se interpretará como el valor '0').

En cambio, si se definen funciones de transformación para el método destino, el filtrado deberá hacerse en función de los parámetros que especifiquen estas funciones, de tal forma que la lista de argumentos construida contenga los especificados por las funciones de transformación. En el atributo *inputMethodParameters* de la clase *WeavingMethod* se almacena esta información. En cambio, el atributo *outputMethodParameters* contiene el formato que tendrá la lista de argumentos del método destino,

sin tener en cuenta las funciones de transformación, para poder posteriormente recuperar los resultados obtenidos. El método *callMethod*, definido en *WeavingMethod*, tomando como entrada una lista de argumentos cuyo formato coincida con el especificado en *inputMethodParameters*, es el encargado de aplicar las distintas funciones de transformación a los argumentos proporcionados y de invocar el método destino con la lista de argumentos adecuadamente transformada.

Por ejemplo, si el método destino necesita como argumentos la lista: *{string, decimal}*, y se define una función de transformación tal que *f(x: int)* → *decimal*, el atributo *inputMethodParameters* contendrá la lista: *{string, int}*, mientras que el atributo o*utputMethodParameters = {string, decimal}*. Suponiendo que en ejecución la lista de parámetros del método origen es {“abba”,49.5, -5, “a”}, el filtrado de dicha lista gracias a la información contenida en *inputMethodParameters* generará la lista {“abba”, -5}, y tras aplicar la función de transformación, podrá invocarse el método destino con los parámetros correctos. Suponiendo que los dos parámetros del método fuesen de retorno y que como resultado ha devuelto {“bb”,2.2}, gracias a la información contenida en *outputMethodParameters* puede filtrarse esta información y depositarse de nuevo sobre la lista de parámetros del método origen en la posición correcta: {“bb”, 49.5, 2.2, “a”}.

♦ La aplicación de los weavings

La aplicación de un weaving es realizada por el hilo del componente tras comprobar previamente que el servicio a ejecutar por el aspecto tiene un weaving asociado, mediante la invocación al método *DoMultiWeaving*. Este método se encarga de buscar en la lista dinámica de métodos a los que se le aplica weaving, y en caso de encontrarlo, va recorriendo las listas *beforeList*, *afterList* e *insteadList* y ejecutando los delegados correspondientes en el orden adecuado. La signatura de dicho método es la siguiente:

```
public AsyncResult DoMultiWeaving(IAspect aspect,
              string origMethodName, object[] args)
```

La aplicación de un weaving supone una ejecución secuencial de los distintos servicios implicados, es decir, del método origen y los distintos métodos destino, cuya invocación es desencadenada por la aplicación del weaving asociado al método origen. Esta ejecución es secuencial debido a que la propia definición de weaving establece un orden de ejecución de los distintos servicios, de lo que se deriva que hasta que un servicio no se haya procesado completamente no podrá procesarse el siguiente.

Para la ejecución de los weavings se consideraron dos posibles estrategias:

- **A través del hilo de ejecución del componente**

  Esta estrategia no permite reenviar las nuevas peticiones a los aspectos hasta que el weaving finalice. En un primer prototipo se aplicó esta estragia, cuya ventaja principal es una mayor sencillez y la conservación de un estado coherente en los aspectos, pues hasta que no ha finalizado un weaving por completo el aspecto no recibe las nuevas peticiones.

  Sin embargo, el inconveniente viene cuando un servicio a ejecutar por el weaving no puede procesarse debido al estado actual del protocolo. En muchos casos la actuación esperada es dejar dicho servicio encolado en el aspecto (en espera, mediante el mecanismo descrito en el apartado de los aspectos) hasta la llegada de otro servicio que cambie el estado del protocolo a uno válido para la ejecución del servicio. No obstante, como el weaving requiere una ejecución síncrona, si el servicio está en espera el hilo del componente se encontrará en estado suspendido a la espera de la finalización del servicio, por lo que no podrá reenviar nuevas peticiones de servicio y por tanto nunca cambiará el estado del aspecto. Se produce por tanto un interbloqueo.

- **A través de un nuevo hilo de ejecución exclusivo**

  Esta estrategia libera al hilo del componente de la ejecución de los weavings, permitiendo que reenvíe nuevas peticiones a los aspectos y por tanto, solucionar el problema descrito anteriormente. También incrementa el rendimiento del componente, pues mientras se está procesando un weaving, pueden reenviarse nuevas peticiones de servicio a otros aspectos.

  Sin embargo, esta estrategia presenta el inconveniente de aumentar la probabilidad de perder el orden de ejecución de los servicios enviados a un aspecto, a causa de la concurrencia. En el caso de estar en un weaving en ejecución, y recibirse una petición de servicio cuyo aspecto estuviese implicado en el weaving, podría ocurrir que la nueva petición de servicio llegase antes al aspecto que por parte del weaving, por lo que el orden de ejecución se vería alterado.

De las dos estrategias descritas, se ha implementado la segunda, aunque el problema presentado no ha sido resuelto en el prototipo actual. Una posible solución al problema sería la creación de unos servicios de espera específicos en los aspectos, de forma que cuando el weaving se iniciase, detectase todos los aspectos relacionados e invocase dicho servicio. Este servicio, cuando fuese procesado por el aspecto, quedaría a la espera hasta que el weaving actualizase su estado, proporcionando el servicio concreto a ejecutar. De esta

forma, las peticiones posteriores al weaving no podrían ser procesadas hasta que el weaving finalizase.

El método *DoMultiWeaving* cuando es invocado crea una instancia de la clase *MultiWeaving*, proporcionándole los argumentos del método origen, el objeto *AsyncResult* donde depositar los resultados y el *WeavingNode* en el que se encuentra toda la información del weaving a aplicar. Esta instancia es la que crea un hilo de ejecución adicional para gestionar cada weaving. Toda la funcionalidad para el procesado de un weaving se encuentra definida en el método *DoWeaving*(), cuya estructura se procederá a describir.

La invocación de un servicio que tiene weaving asociado (método origen) desencadenará la ejecución de distintos servicios (métodos destino) en un orden definido por el weaving. Tras lanzarse a ejecución el método *DoWeaving*, en primer lugar se van ejecutando los distintos métodos destino almacenados en la lista *WeavingNode.afterList* (el método origen se ejecutará después de los definidos en esta lista). En segundo lugar se comprueba si existen métodos destino en la lista *WeavingNode.insteadList*, en cuyo caso se ejecutan secuencialmente, sustituyendo la ejecución del método origen. En caso negativo, se ejecuta el método origen. En tercer lugar, se ejecutan los distintos métodos almacenados en la lista *WeavingNode.beforeList*.

A continuación se muestra un extracto del código del método *DoWeaving*, el correspondiente a la ejecución de los weaving *after* y *afterIf*:

```
AsyncResult result = null;
object[] args = this.args;
WeavingNode nodo = this.node;

// Si hay un weaving AFTERIF y su condición no se evalúa a cierto, no se
// ejecutará el método origen
bool conditionalExecution = true;

// Obtenemos los parámetros del nodo origen
ParameterInfo[] origMethodParameters = nodo.Method.Method.GetParameters();

// Realizamos el weaving AFTER
if (nodo.AfterList != null && nodo.AfterList.Count > 0) {
   foreach (WeavingMethod nodoAfter in nodo.AfterList) {
      try {
         result = (AsyncResult)
              nodoAfter.CallMethod(ParameterMatching(origMethodParameters,
                    args, nodoAfter.InputMethodParameters,null));
         // Como el resultado del delegado anterior puede influir en el
         // siguiente, debemos esperar a que éste acabe (suspendido), tras lo
         // cual seremos despertados por el aspecto
         if (!result.HasResult())
            System.Threading.Thread.CurrentThread.Suspend();
      } catch (TargetInvocationException e) {
         Console.WriteLine(e.StackTrace); throw e.InnerException;  }

      // Obtenemos los resultados de la llamada síncrona.
      if (result.HasResult()) {
         object[] oldArgs = result.GetParameters();
         args = ParameterMatching(nodoAfter.OutPutMethodParameters, oldArgs,
                   origMethodParameters, args);
```

```
            // Comprobamos si se ha de ejecutar el método origen
            if (conditionalExecution)
               conditionalExecution = nodoAfter.CheckConditionalWeaving(oldArgs);
         }
         else {
            throw new Exception("Critical Error in Threading section");
         }
      }
   }

   // Si no hay miembros en la lista de instead, ejecutamos el METODO ORIGEN
   if (nodo.InsteadList == null || nodo.InsteadList.Count == 0) {
      // Comprobamos si había una clausula AFTERIF con una condicion
      if (conditionalExecution) {

   // ....
```

Como se puede observar, a través de un bucle se van procesando todos los weaving *after* (y *afterIf,* pues se almacenan en la misma lista) de forma secuencial. En primer lugar, mediante la invocación al método *ParameterMatching* se filtra la lista de argumentos del método origen generando la lista que espera recibir el método destino correspondiente. En segundo lugar, el método destino es invocado a través del método *CallMethod*, definido en la clase *WeavingMethod*, el cual aplica las funciones de transformación que hubiesen definidas. Tras esto, el hilo del weaving queda suspendido a la espera que la invocación del servicio finalice. Una vez despertado el hilo del weaving, obtiene los resultados de la estructura *AsyncResult*, y comprueba si se trataba de un weaving condicional (*afterIf)*, invocando para ello el método *CheckConditionalWeaving()* también definido en la clase *WeavingMethod*. Finalmente, se filtran de nuevo los argumentos para que se ajusten a la signatura del método origen.

De esta forma, la creación de weavings es totalmente dinámica, sin más que especificar los aspectos a entretejer, los nombres de los métodos correspondientes y el tipo de weaving a aplicar. El método *AddWeaving* es el encargado, a través de la reflexión, de obtener la información de ambos métodos, crear dinámicamente los delegados y almacenar los datos adecuadamente en la estructura *WeavingsCollection*. Sin embargo, la creación de delegados en tiempo de ejecución no ha sido una tarea sencilla. La forma común de crear delegados es definiendo el tipo en tiempo de compilación e instanciándolo en tiempo de ejecución. Pero si los aspectos son agregados en tiempo de ejecución, el componente no conoce el tipo de los delegados necesarios para acceder a sus métodos, por lo que no puede instanciarlos.

Para solucionar este problema, pueden seguirse dos estrategias. La primera de ellas es la que se utilizó en un primer prototipo, y se basa en la creación de delegados de forma dinámica emitiendo código. Aunque en .NET existe una clase de la cual hereda todo delegado (las clases Delegate y MulticastDelegate), no existe ningún constructor para crear dinámicamente delegados, por lo que hay que recurrir a emitir código. El proceso a seguir es

generar el código intermedio correspondiente al tipo de delegado requerido, creándose un ensamblado en memoria, y posteriormente instanciar dicho delegado para que apunte al método requerido. El principal inconveniente, aparte de que debe emitirse código para cada delegado, es que si el componente es movido a otra máquina debe volverse a generar el delegado.

Otra alternativa posible al uso de los delegados sería hacer uso de la reflexión y el método *Invoke()* para invocar a los métodos, pero el rendimiento obtenido es inferior, como se demuestra en [Gun04], donde se puede encontrar un estudio sobre las distintas formas de invocar código dinámicamente en .NET. No obstante, en la versión de .NET *Framework* 2.0 han mejorado el rendimiento obtenido mediante el uso de la reflexión y han añadido mecanismos para crear dinámicamente delegados.

La segunda estrategia presupone que los tipos de delegados han sido definidos junto a las interfaces que definen los aspectos, por lo que para instanciar un delegado no hay más que buscar el tipo requerido en el ensamblado correspondiente. Esta estrategia es la que finalmente se ha implementado, puesto que los puertos también hacen uso de los delegados para encolar las peticiones de servicio. En la clase *Delegates* se encuentra la implementación de las dos estrategias.

4.2.4.4 Puertos

Los puertos son entidades agregadas al componente que, además de proporcionar los mecanismos necesarios para comunicarse con otros elementos arquitectónicos, facilitan la evolución de los componentes. El componente por sí solo (como instancia de la clase *ComponentBase*) únicamente proporciona el comportamiento para gestionar weavings y aspectos, y para redirigir servicios a los aspectos. Éstos, al ser agregados al componente, proporcionan la implementación de los servicios definidos en una serie de interfaces, y a través de los puertos, cada una de las interfaces que implementan los aspectos es publicada al exterior. Cuando un aspecto es eliminado de un componente, también son eliminados los puertos asociados, con lo que el componente deja de publicar los servicios asociados al aspecto. Esto se ha realizado de esta forma con dos objetivos: por una parte, que el componente sea totalmente independiente de las partes que lo componen, favoreciendo con ello la reutilización, y por otra parte, que la agregación/eliminación de aspectos, weavings o puertos pueda realizarse en tiempo de ejecución sin necesidad de recompilar el componente. Solamente deberá pararse el componente en aquellos casos en los que la modificación de alguna de sus partes afecte a un servicio en ejecución.

Los puertos también ofrecen mecanismos de reflexión, ya que pueden ser interrogados dinámicamente para obtener la lista de interfaces que publica el componente al cual están agregados, posibilitando comunicaciones totalmente dinámicas. Además, los puertos permiten aislar los servicios que

proporcionan los aspectos (los especificados por el desarrollador) de los métodos que publican los componentes para su manejo por parte del *middleware*. Por ejemplo, podría darse el caso de que en los aspectos también se definiesen servicios cuyo interfaz fuese *Start()* y *Stop*(), que entrarían en conflicto con los heredados de *ComponentBase*. También proporcionan una solución para el problema de idénticos servicios en distintas interfaces, al definirse un puerto diferente para cada interfaz.

Todo puerto hereda de una clase padre *Port* (ver Figura 33), que define los atributos básicos que tienen los puertos:

- **portName**: nombre identificativo del puerto, tal como ha sido definido en la especificación PRISMA de un componente.
- **interfaceName**: nombre de la interfaz que implementa el puerto.
- **subProcess**: nombre del subproceso vinculado al puerto.
- **attachmentRegistered**: es un vector que contiene información sobre los attachments que están comunicándose con el puerto, con el objetivo de que si el componente es movido, o detenido, el componente pueda notificar al *middleware* a qué attachments tiene que avisar.

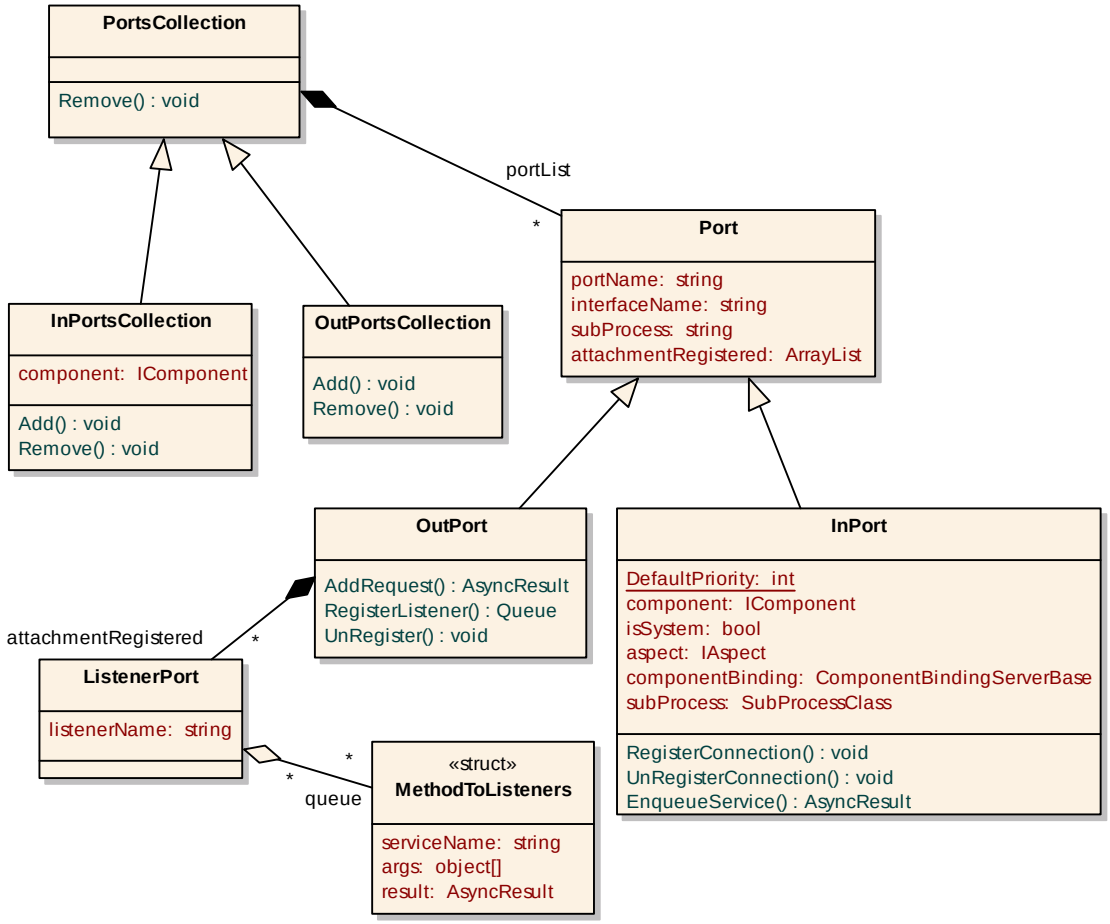

Figura 33 - Clases InPort, OutPort y ListenerPort

En implementación, los puertos se clasifican, por su función, en puertos de entrada y puertos de salida. Los de entrada publican los servicios que ofrece el componente y reciben las peticiones de otros componentes. Mientras que los de salida se encargan de entregar las peticiones generadas por el componente a los componentes externos a los cuales esté conectado.

Los puertos se agregan al componente a través de instancias de las clases *InPortsCollection* y *OutPortsCollection*, para los puertos de entrada y los puertos de salida, respectivamente. Estas clases son las encargadas de gestionar los puertos agregados al componente. Ambas heredan de la clase *PortsCollection*, que define como atributo básico, una lista dinámica (*portList*) y la signatura de los métodos para eliminar los puertos de dicha lista. La implementación de los métodos para añadir o eliminar puertos a dicha colección es realizada por sus descendientes, pues los puertos de entrada requieren parámetros de inicialización distintos a los de salida. Los métodos que permiten eliminar puertos tienen la siguiente signatura:

```
void Remove(string interfaceName, string subProcesses);
void Remove(string portName);
```

Las dos sobrecargas del método permiten eliminar un puerto, tanto de entrada como de salida, a partir de su nombre identificativo o por la interfaz y el nombre del subproceso (*played_role*) que lo identifica.

La clase *InPortsCollection* es la encargada de gestionar la lista de puertos de entrada. Proporciona dos *indexers* para poder acceder a los puertos por nombre o por el par [nombre_interfaz, nombre_subproceso]. Ofrece la implementación de los métodos *Add* y *Remove*. La signatura del método *Add* se muestra a continuación:

```
void Add(string portName, string interfaceName,
              IAspect implementerAspect, string SubProcessName)
```

El método *Add* crea un puerto de entrada a partir del nombre de la interfaz que debe implementar, dándole un nombre identificativo (*portName*), la referencia al aspecto que implementa y el nombre del subproceso que representa (si es aplicable).

De forma similar, la clase *OutPortsCollection* se encarga de gestionar la lista de puertos de salida. Proporciona también dos *indexers* para obtener los puertos por el nombre o por la interfaz+subproceso. También proporciona la implementación para los métodos *Add* y *Remove*. A diferencia de los puertos de entrada, los de salida no requieren un aspecto asociado, por lo que la signatura del método *Add* es distinta:

```
void Add(string portName, string interfaceName, string subProcessName)
```

♦ Puertos de entrada

El comportamiento de los puertos de entrada se define en la clase *InPort*, que a su vez hereda de la clase *Port*. Un puerto de entrada, desde el momento de su creación, está asociado al aspecto el cual implementa los servicios que él publica. Estos servicios pueden tener una prioridad asignada, que vendrá determinada por los subprocesos (*played_roles*) del aspecto. Los atributos y servicios que define esta clase son:

- **DefaultPriority**: atributo que define la prioridad por defecto para aquellos servicios que no tengan prioridad asignada.
- **component**: atributo que guarda la referencia hacia el componente al cual está agregado el puerto. Servirá para poder enviar las peticiones recibidas.
- **isSystem**: flag que permite conocer si el puerto está asociado a un sistema, en cuyo caso el funcionamiento es ligeramente distinto, pues puede que el puerto también tenga asociado un binding al que redirigirle la petición.
- **aspect**: atributo que guarda la referencia hacia el aspecto al cual está asociado el puerto. Es necesario para poder instanciar los delegados.
- **componentBinding**: atributo que almacena el binding al cual redirigir la petición de servicio recibida, en el caso de ser un puerto perteneciente a un sistema.
- **subProcess**: atributo que almacena la información relacionada con los subprocesos (*played_roles*) del aspecto y permite obtener la prioridad de cada servicio.
- **RegisterConnection()** y **UnRegisterConnection()**: servicios que almacenan o borran, respectivamente, el nombre del attachment que se está comunicando con este puerto en el vector heredado *attachmentRegistered*.
- **EnqueueService()**: método protegido que sólo podrá ser invocado por las clases que hereden de *InPort*. Se encarga de almacenar en la cola del componente o del sistema, según sea el caso, la petición de servicio, que consiste en un delegado y un vector con sus argumentos. También encapsula en un sólo método las diferencias entre puertos de sistemas y componentes.

De forma similar a los aspectos, la implementación de los puertos de entrada se encuentra separada, por una parte en el comportamiento básico, y por otra parte en la definición de la interfaz que publica el puerto. Mientras que la clase *InPort* define cómo el puerto de entrada se comporta e interactúa con el componente, la especificación de los servicios de las interfaces que publica el puerto se implementan en otras clases descendientes de *InPort*. De esta forma, cuando se compila una interfaz PRISMA, también se genera el puerto de entrada correspondiente que

publicará dicha interfaz. Cuando el puerto de entrada es agregado al componente, se carga el tipo de puerto correspondiente a la interfaz requerida y se instancia con los valores iniciales necesarios. La convención utilizada para que el *middleware* sea capaz de encontrar los puertos requeridos para cada interfaz es nombrar a los tipos generados como "*InPort*"+nombre_interfaz.

Para los componentes externos, acceder a un servicio publicado por un componente, se traduce a una línea de código bastante intuitiva:

```
Nombre_componente.InPorts[Nombre_interfaz].NombreServicio(argumentos);
```

A continuación se muestra un ejemplo de la sentencia que se ha de incluir en el código para que una instancia del componente *Account* accediese al servicio *Withdrawal*, cuya interfaz es *ICreditCardTransactions*.

```
Account1.InPorts["ICreditCardTransactions"].Withdrawal(qtity,money);
```

La plantilla de código que debería generar el compilador para especificar un puerto de entrada que publicase servicios de la interfaz *ICreditCardTransactions* sería la siguiente:

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Components;
using PRISMA.Components.Ports;

namespace CuentaBancaria {
   // Esta clase implementa el puerto de entrada correspondiente a la interfaz
   // "ICreditCardTransactions". Los componentes que tengan dicha interfaz como
   // servicio, usarán este puerto.
   [Serializable]
   public class InPortICreditCardTransactions: InPort,ICreditCardTransactions {
      // Constructor del InPort. Proporciona al padre los parámetros necesarios
      public InPortICreditCardTransactions(string inPortName,
         string subProcessName, IAspect aspect, IComponent component) :
            base(inPortName,"ICreditCardTransactions", subProcessName,aspect,
                  component) {}

      #region Servicios ofrecidos por ICreditCardTransactions
      // Aquí se define cada uno de los servicios que publica la interfaz

      public AsyncResult Withdrawal(decimal quantity, ref decimal money) {
         lock(this) {
            // Creamos el delegado hacia el servicio correspondiente
            if (aspect!=null)
               withdrawaldelegate = new WithdrawalDelegate(
                  ((ICreditCardTransactions)this.aspect).Withdrawal);
            money = 0;

            // Encolamos la petición
            return (AsyncResult) this.EnqueueService("Withdrawal",
                  withdrawaldelegate, quantity, money);
         }
      }
      // ...
      #endregion
   }
}
```

Como se puede observar, el código a generar es mínimo, pues la mayor parte del comportamiento es heredado de la clase *InPort*. Al igual que las demás entidades implementadas, los puertos de entrada deben ser etiquetados con el atributo *Serializable* para permitir que puedan moverse con el componente. El constructor del puerto únicamente requiere los argumentos declarados por el constructor del padre, como el nombre del puerto, el aspecto que implementa la interfaz, el subproceso (*played_role*) que representa y una referencia al componente. Por cada servicio especificado en la interfaz debe definirse un servicio con la misma signatura que, tras bloquear el puerto para permitir un acceso concurrente seguro, cree el delegado hacia el servicio del aspecto que lo implementa, y lo encole en el componente a través del método heredado *InPort.EnqueueService*.

- **Puertos de salida**

Los puertos de salida se han diseñado para permitir la petición de servicios por parte de los aspectos de un componente a los demás elementos arquitectónicos, sin que para ello los aspectos deban conocer el elemento arquitectónico concreto con el que se están comunicando, y favorecer con ello la reutilización. Esto es posible gracias a que los componentes y conectores son conectados entre sí a través de los attachments, que son los encargados de recoger las peticiones depositadas por los aspectos en los puertos de salida y reenviarlas a los puertos de entrada del otro extremo.

Un puerto de salida se encuentra implementado en la clase *OutPort*, y viene identificado por el nombre del puerto o bien por la interfaz que implementa y el subproceso (*playedRole*) que representa, de la misma forma que un puerto de entrada. Cuando los aspectos desean solicitar un servicio de otro componente, depositan sus peticiones en el puerto de salida. Éstas se almacenan en una cola, con la finalidad de que sean procesadas en el mismo orden que fueron enviadas. Sin embargo, como puede que varios attachments estén pendientes de un mismo puerto de salida, aparece el problema de quién o cuándo debe borrarse de la cola la petición de servicio. Si es borrada demasiado pronto, puede que algún attachment no hubiese recogido la petición. Si es desencolada demasiado tarde, puede que sea recogida por un attachment que ya la había recogido previamente. El problema se ha solucionado creando un mecanismo de registro en los puertos de salida que ofrece a cada suscriptor (los attachments) un canal exclusivo en el que se depositarán las peticiones a él dirigidas. Como el suscriptor será el único que accederá a ese canal, podrá procesarlas a su ritmo de proceso, establecido por la latencia de las redes que tenga que atravesar.

Las peticiones de servicio son depositadas en el puerto mediante una invocación al método *AddRequest(method: string, args: object[])*. El aspecto únicamente proporciona el nombre del método y los argumentos necesarios,

siendo los attachments los encargados de traspasar la petición al servicio adecuado.

A continuación se muestra el aspecto de coordinación del apartado 3.4.1.2, en el que se define el protocolo a seguir cuando se reciba una petición *Withdrawal*:

```
COORD = (CLIENT.Withdrawal?(quantity, money) →
        SERVER.Withdrawal!(quantity, money) →
        SERVER.Withdrawal?(quantity, money) →
        CLIENT.Withdrawal!(quantity, money)).COORD
```

Este protocolo especifica que cuando se reciba una petición del servicio *Withdrawal* por el puerto cuyo *played_role* es CLIENT, deberá reenviarse dicha petición al puerto cuyo *played_role* es SERVER. Sin embargo, en la especificación de los aspectos PRISMA no se hace mención alguna a los puertos del componente a los cuales el aspecto puede solicitar peticiones de servicio externas. Esto es debido a que el aspecto no conoce los puertos que formarán parte del componente, con el objetivo de favorecer la reutilización de aspectos y componentes. Por este motivo, en la especificación únicamente se indica que un servicio se solicitará a una interfaz con un *played_role* específico. En la especificación de un componente, los puertos tienen un nombre identificativo, la interfaz y *played_role*, permitiendo establecer una relación entre los puertos del componente y los *played_roles* del aspecto.

En la implementación, esto se traduce en que cuando el método privado del aspecto *_Withdrawal* es invocado por un puerto cuyo subproceso (*played_role*) representado es CLIENT[11], debe invocar el mismo servicio por el puerto cuyo subproceso es SERVER. En el siguiente fragmento de código se muestra cómo se invocaría desde un aspecto al servicio *Withdrawal* por el puerto cuyo *played_role* es SERVER.

```
Object[] args = new object[] {quantity, money}
AsyncResult result =
     link.OutPorts["ICreditCardTransactions","SERVER"]).AddRequest(
     "Withdrawal", args);
```

El atributo *link* de los aspectos es la referencia que éstos tienen hacia el componente que los agrega, mediante la cual pueden acceder al atributo *OutPorts* que contiene la lista de puertos de salida que forman parte del componente. El aspecto obtiene la referencia del puerto en el que depositar la petición de un servicio a través de la interfaz que publica y el *played_role* que representa. Finalmente, la invocación del método *AddRequest* devolverá un objeto *AsyncResult* para procesar posteriormente los resultados obtenidos.

Una vez añadida una petición de servicio al puerto de salida, ésta debe ser proporcionada a los attachments que están suscritos al puerto. Mediante los

[11] La identificación del subproceso para servicios entrantes, es decir, proporcionados por el puerto de entrada, no ha sido implementada en el prototipo actual. En cambio, sí que ha sido implementada para los servicios salientes.

métodos *RegisterListener(name: string)* y *UnRegister(name:string)* los attachments pueden suscribirse o desuscribirse a un puerto de salida y obtener con ello el canal exclusivo, que consiste en una referencia a una instancia de la clase *ListenerPort*. Una instancia de esta clase es creada cuando el attachment se suscribe al puerto, borrada cuando se desuscribe y se almacena en la lista heredada *Port.attachmentsRegistered*. Contiene el nombre del suscriptor y una cola en la que dejar las peticiones destinadas a dicho suscriptor. Por tanto, cuando al puerto de salida le llega una nueva petición de servicio (como instancia de *MethodToListener*), ésta será copiada en las colas de los objetos *ListenerPort* asociadas a cada suscriptor. Periódicamente, como se verá en apartados posteriores, el suscriptor consultará dicha cola para comprobar si tiene peticiones pendientes de procesar.

4.2.4.5 Definición de Componentes y Conectores en .NET

Una vez vistos todos los elementos que forman parte de un componente, en esta sección se verá de forma global cómo se combinan para representar un componente especificado en PRISMA. A continuación se muestra el código correspondiente a la implementación del componente *Account*, cuya especificación PRISMA ha sido mostrada en el apartado 3.4.1.3.

```
using System;
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace CuentaBancaria {
   // Definición del Componente ACCOUNT
   [Serializable]
   public class Account: ComponentBase {

      // Constructor del componente Account
      public Account(string AccountName,
            <Atributos_requeridos_por_los_aspectos>
            decimal accountId, LOC initialLocation,
            <\Atributos_requeridos_por_los_aspectos>
            MiddlewareSystem middlewareSystem) :
               base (AccountName, middlewareSystem) {

         <Definicion_Aspectos>
         // Creación del aspecto funcional
         IAspect functionalAspect = new BankInteraction(accountId);
         AddAspect(functionalAspect);
         // Creación del aspecto distribución
         IAspect distributionAspect = new ExtMbile(initialLocation);
         AddAspect(distributionAspect);
         <\Definicion_Aspectos>

         /** Otra forma de obtener las referencias a los aspectos agregados: **
         IAspect functionalAspect = GetAspect(typeof(FunctionalAspect));
         IAspect distributionAspect = GetAspect(typeof(DistributionAspect));
         ***********************************************************************/

         <Definicion_Weavings>
```

```
        // Creamos el Weaving entre el aspecto funcional y el de distribución
        AddWeaving( functionalAspect, "ChangeAddress",
                  WeavingType.BEFORE,
                distributionAspect, "Move");
        <\Definicion_Weavings>

        <Definicion_Puertos>
        InPorts.Add("AccountPort","ICreditCardTransactions",
             functionalAspect, "SERVER");
        OutPorts.Add("AccountPort","ICreditCardTransactions","SERVER");
        <\Definicion_Puertos>
      }
   }
}
```

Como se puede observar, todo el código correspondiente a la definición de las propiedades del componente (aspectos, weavings y puertos) se define en el cuerpo del constructor de la clase. La clase *Account* representa al componente del mismo nombre especificado en PRISMA, y hereda de *ComponentBase* todo su comportamiento, como ya se ha descrito en las secciones anteriores. Si se estuviese definiendo un conector, la clase también debería especificar que implementa la interfaz *IConnector*, con el objetivo de que sea tratado como tal.

Los parámetros iniciales que requiere el constructor son el nombre identificativo que tendrá la instancia del componente creado, una referencia al *middleware*, y los atributos iniciales que necesiten los aspectos, tal como se define en la sección *Initialize* del ADL PRISMA. El cuerpo del constructor sigue una estructura genérica:

- *Definición de los Aspectos*: en primer lugar se crean los aspectos, se almacenan en una variable local para facilitar posteriormente su acceso, y se agregan al componente a través de invocaciones al método *AddAspect*.
- *Definición de los Weavings*: en segundo lugar se crean los weavings mediante invocaciones al método *AddWeaving*, tal como se vió en el apartado correspondiente.
- *Definición de los Puertos*: finalmente se crean los puertos de entrada y de salida, mediante las invocaciones respectivas a los métodos *Add*, con lo que las interfaces implementadas por los aspectos son publicadas al exterior y se habilita el medio de comunicación de los mismos hacia otros componentes externos.

El orden en que se realizan los procesos es importante, pues para crear un puerto o un weaving es necesario que previamente se haya agregado el aspecto correspondiente.

De este modo el componente, gracias a los distintos módulos que lo integran, permite la modificación de su estructura interna (puertos, aspectos y weavings) en tiempo de ejecución sin necesidad de detener sus operaciones por completo y recompilarse. Sin embargo, el prototipo actual únicamente

proporciona los servicios necesarios para que cada instancia de un componente, a modo individual, pueda ser modificada. En cambio, para que el prototipo soporte la evolución de forma completa aún tienen que ser introducidos una serie de mecanismos adicionales que permitan la coordinación entre las diferentes instancias y la propagación de los cambios a los distintos *middlewares*.

4.2.5 Sistemas

Una vez vista la implementación de los componentes PRISMA, a continuación se presentará el diseño y la implementación de los sistemas. Estas entidades permiten especificar componentes complejos, formados a partir de la agregación de componentes y conectores, interconectados entre sí mediante attachments, y conectados a los puertos del sistema a través de bindings.

4.2.5.1 Modelo de ejecución

Un sistema es un componente que puede tener aspectos, definir weavings entre ellos y publicar los servicios implementados por los aspectos a través de los puertos. Además, al ser una componente compleja, puede estar compuesto a su vez por componentes y conectores y definir bindings y attachments. Por tanto, un sistema es un componente que extiende su comportamiento con servicios adicionales para gestionar las relaciones de composición entre los elementos que lo forman.

Es por esto que el sistema se ha implementado como una clase especializada de *ComponentBase*, por lo que hereda el comportamiento de los componentes, pero que además define una serie de servicios específicos que caracterizan al sistema, agrupados en la interfaz *ISystem* (ver Figura 34). Dichos servicios permiten añadir o eliminar componentes (*AddComponent*, *RemoveComponent*), conectores (*AddConnector*, *RemoveConnector*), attachments (*AddAttachment, RemoveAttachment*) o bindings (*AddBinding*, *RemoveBinding*). La clase *SystemBase* es la que implementa esta interfaz.

Todas las entidades que forman parte de un sistema (componentes, conectores, attachments y bindings) son instanciadas, mantenidas y eliminadas por el *middleware*. El sistema únicamente comprueba que se cumplen las precondiciones (que el componente a agregar no ha sido definido ya, que un attachment se define para un componente y conector, etc.) e invoca los servicios correspondientes del *middleware*. El *middleware* se encarga de buscar el tipo necesario, cargarlo y ponerlo en ejecución (para los servicios de creación), de mantener las distintas entidades en un estado consistente (para los servicios de movilidad), y de eliminarlas. El sistema únicamente mantiene una serie de listas dinámicas con los nombres identificativos (y únicos) de las distintas entidades que lo forman. En tiempo

de ejecución, cuando el sistema requiera ejecutar un servicio del *middleware* relacionado con alguna de las entidades que lo forman (por ejemplo, mover un componente, eliminar un attachment, etc.), proporcionará su correspondiente identificador.

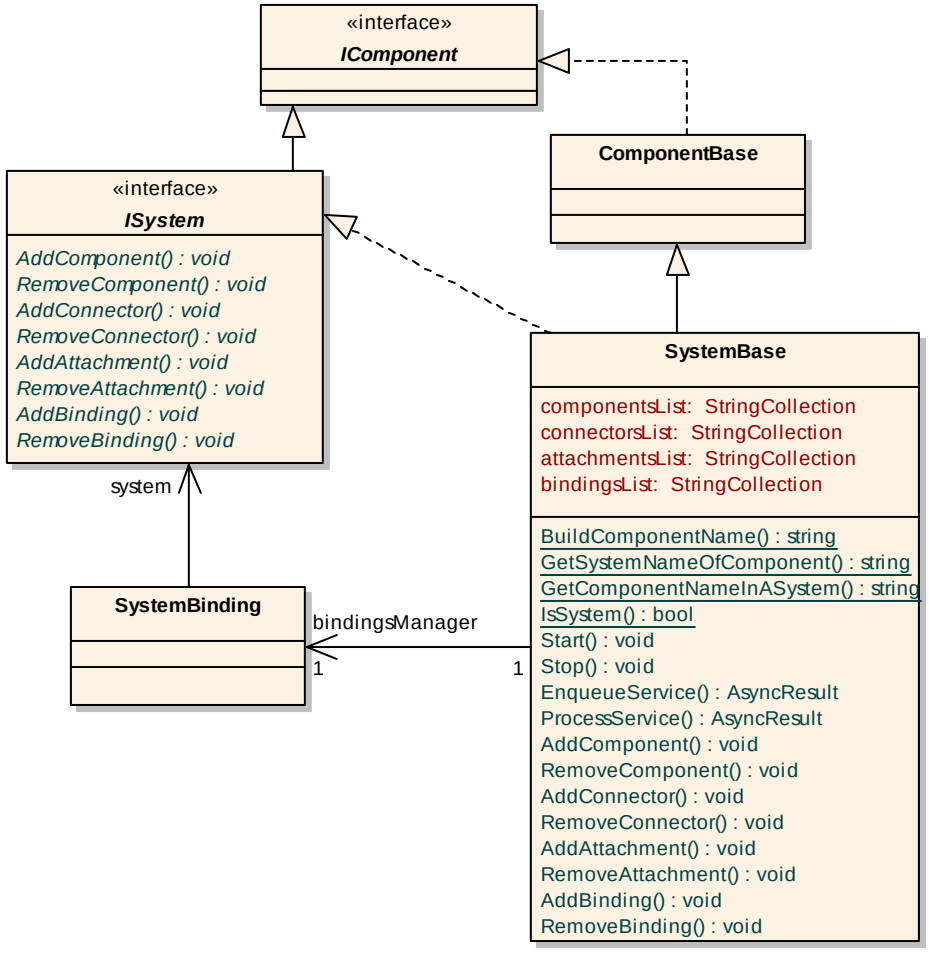

Figura 34 - Interfaz ISystem y clase SystemBase

En un diseño preliminar del Sistema, se intentó que las entidades que lo forman (componentes, conectores, attachments y bindings) estuviesen encapsuladas en su interior como objetos agregados, en lugar de guardar sólo el identificador. También se planteó la posibilidad de que la gestión (creación, modificación y borrado) de las distintas entidades fuese realizada completamente por el sistema. Sin embargo, en el caso de aquellos componentes y conectores internos de un sistema que residan en máquinas distintas, la gestión sólo podrá realizarse en la máquina correspondiente donde se alojen. Además, el sistema no podría tener la referencia remota de dichos elementos ya que, como se verá más adelante, los componentes y conectores no se publican para acceso remoto. En cambio, si la gestión es realizada parcialmente por el *middleware*, éste puede encargarse de propagar la solicitud de determinados servicios (i.e. mover un componente a otra ubicación) al *middleware* que contenga el componente remoto, de forma totalmente transparente para el sistema. Es preferible que el sistema se

ocupe de los requisitos funcionales (mayor nivel de abstracción) y dejar que el *middleware* realice las operaciones de más bajo nivel, como la creación y eliminación de objetos, acceso remoto, serialización, búsqueda y transferencia de tipos, etc.

Por esta razón, la clase *SystemBase* define los atributos *componentsList*, *connectorsList*, *attachmentsList* y *bindingsList* como listas dinámicas de cadenas, en las que se almacenan los nombres identificadores de cada uno de los elementos que forman parte del sistema. Es fundamental que dichos nombres sean únicos para todo modelo arquitectónico PRISMA en ejecución, para que el *middleware* pueda localizar de forma unívoca dónde se encuentran los elementos residentes en distintas máquinas. A nivel de implementación los componentes/conectores son tratados por el *middleware* como si estuviesen agregados de forma referencial. Esto es necesario para que el *middleware* pueda gestionar cada elemento de forma individual y, además de su creación y destrucción, poder moverlos a otras máquinas independientemente de la máquina donde se encuentre el sistema. En caso contrario, los componentes y conectores no podrían ser reubicados a distintas máquinas, pues el *middleware* no tendría acceso a ellos.

En la especificación PRISMA de un sistema, para cada componente y conector agregado, se asocia un nombre identificativo mediante el cual se puede hacer referencia a dichos componentes internos dentro del ámbito del sistema. Sin embargo, cuando un sistema es instanciado, debe utilizarse un nombre que identifique de forma única a cada componente interno agregado al sistema, y que sea válido a nivel de modelo arquitectónico. Esto es necesario debido a que cada componente interno es registrado en el *middleware* como si fuese un componente independiente. Por ejemplo, si se define un sistema del tipo "SystemType" formado por un componente cuyo nombre es "Component", y se crean dos instancias de dicho sistema, "System1" y "System2", los componentes internos no podrían registrarse en el *middleware* como "Component", pues no habría forma de distinguir cuál de los dos pertenece a cada sistema.

Para solucionar este problema, se ha establecido que el nombre identificativo de un componente interno se forme por la concatenación del nombre de la instancia del sistema al cual pertenece, seguido del nombre del componente interno definido en la especificación. De esta forma, los nombres de los componentes internos del ejemplo anterior serían: "System1.Component" y "System2.Component". El método estático *BuildComponentName* construye el nombre identificativo de un componente o conector interno tomando como entrada el nombre del sistema y el nombre del componente definido en la especificación. Los métodos estáticos *GetSystemNameOfComponent* y *GetComponentNameInASystem* permiten obtener a partir del nombre identificativo de un componente interno, el nombre del sistema al cual pertenece y el nombre del componente definido en la especificación, respectivamente.

Por otra parte, el método estático *IsSystem* permite, dado un tipo, comprobar si es un sistema. Es utilizado por el *middleware* en la carga de tipos. Los métodos de creación y borrado de componentes, conectores, attachments y bindings, de forma general, comprueban en primer lugar si se cumplen las distintas precondiciones, como que el elemento a crear no existe ya o que los tipos proporcionados como parámetros son los correctos. Después, invocan los servicios equivalentes del *middleware* (*CreateComponent, RemoveComponent, CreateAttachment, RemoveAttachment*) o del gestor de bindings (*CreateBinding*, *RemoveBinding*). Finalmente, si dicha invocación ha tenido éxito, los nombres identificativos de los elementos creados (o eliminados) son añadidos (o borrados) de las listas internas del sistema. A continuación se muestran las signaturas de los distintos métodos que permiten modificar los elementos arquitectónicos que componen el sistema:

```
void AddComponent(string componentName, Type componentType, string targetURL,
                  object[] args);
void RemoveComponent(string componentName);

void AddConnector(string connectorName, Type connectorType, string targetURL,
                  object[] args);
void RemoveConnector(string connectorName);
```

Como se puede observar, los métodos para crear un componente o un conector requieren los mismos tipos de parámetros, pues su implementación es idéntica, como ya se ha visto anteriormente. Los parámetros requeridos son el nombre definido en la especificación, el tipo del componente/conector a agregar al sistema, la URL donde se ubicará dicho componente/conector inicialmente, y un vector con los parámetros adicionales que pueda requerir. El borrado de un componente o conector se realiza proporcionando el nombre identificativo.

```
void AddAttachment(string componentName, string portName,
     string connectorName, string roleName);
void RemoveAttachment(string attachmentName);
```

La creación de attachments entre un componente y un conector se realiza de forma similar a como se define en la especificación PRISMA. En primer lugar se proporciona el nombre y el puerto del componente que se desea conectar, y en segundo lugar el nombre y el rol del conector. El borrado se realiza proporcionando el nombre del attachment a borrar.

```
void AddBinding(string systemPortName, string componentName,
     string componentPortName);
void RemoveBinding(string bindingName);
```

La creación de bindings se realiza proporcionando el nombre del puerto del sistema que se desea conectar con el componente o conector, el nombre del componente (o conector) y el puerto (o rol). Para borrar un binding debe proporcionarse el nombre identificativo del binding a eliminar.

Debido a que el sistema agrega una serie de elementos adicionales frente a los definidos por *ComponentBase*, la clase *SystemBase* reemplaza los métodos heredados *Start(), Stop()*, *EnqueueService()* y *ProcessService()* para definir el comportamiento adicional. Los métodos *Start* y *Stop* se encargan de indicar al *middleware* que inicie (o detenga) la ejecución de los componentes, conectores, attachments y bindings que lo componen. Tras esto, el sistema invoca el comportamiento heredado *Start()* o *Stop()* para que el hilo de ejecución propio del sistema sea iniciado (o detenido).

Los métodos *EnqueueService* y *ProcessService* han sido reescritos para contemplar la redirección de peticiones de servicio desde los puertos a los bindings asociados. Para el caso en que a un puerto se le ha asociado un binding, el método *EnqueueService* es sobreescrito para, además de insertar en la cola la petición de servicio recibida, añadir también un delegado hacia el binding que debe ejecutarse. Por su parte, el método *ProcessService* añade el comportamiento necesario para invocar adecuadamente dicho delegado.

4.2.5.2 Attachments

En el modelo PRISMA, la comunicación entre componentes y conectores se realiza a través de los attachments. Cada attachment conecta dos elementos (componente y conector) y se encarga de llevar peticiones desde el uno al otro, actuando como si fuera un canal de comunicación. De esta forma, los attachments añaden una capa de abstracción adicional, permitiendo que las comunicaciones distribuidas sean transparentes para componentes y conectores. La comunicación es bidireccional, ya que por una parte se encargan de hacer accesible al componente desde otras ubicaciones remotas (comportamiento servidor), y por otra, se encargan de establecer la comunicación hacia los componentes remotos (comportamiento cliente). Además, también permiten que los distintos elementos arquitectónicos interconectados funcionen de forma independiente, sin que se conozcan entre sí.

A nivel de implementación, la comunicación distribuida entre objetos se ha realizado mediante la tecnología .NET REMOTING, ya que proporciona implementados la mayor parte de los mecanismos para acceder remotamente a objetos así como para serializarlos y moverlos de una máquina a otra. REMOTING permite que una clase que herede de *MarshalByRefObject* sea accesible remotamente, creando automáticamente en el cliente un proxy del objeto remoto que se encargue de establecer las comunicaciones entre cliente y servidor de forma transparente para ambos. Para que un objeto pueda ser accesible remotamente debe ser publicado previamente. Esto se reduce a una llamada al método *RemotingServices. Marshal()*, para el que se le debe proporcionar la instancia del objeto a publicar, el URI a través del cual será accesible, y el tipo. Por otra parte, mediante la serialización un objeto puede ser convertido a un flujo de bytes, enviado por el canal de comunicación y al ser deserializado vuelto a

convertir en un objeto que mantiene su estado previo. Sin embargo, como limitación un objeto que herede de *MarshalByRefObject* siempre será accedido de forma remota y no podrá enviar una copia de sí mismo a otra máquina, porque lo que envíe será siempre un proxy hacia él mismo. Por tanto, un componente PRISMA no podrá tener la capacidad de moverse a la vez que también es accesible remotamente. La movilidad es la capacidad de un componente para moverse a otro equipo y seguir su proceso de ejecución en el punto que lo dejó, manteniendo su estado. Por esta razón, se han separado las propiedades de movilidad y acceso remoto: los componentes conservan la movilidad y los attachments se encargan del acceso remoto. De esta forma, todas las clases que forman parte de los componentes son serializables, mientras que los attachments que deban ser accesibles remotamente heredan de *MarshalByRefObject*.

De forma general, un attachment comunica un puerto de salida de un componente con el puerto de entrada de un conector y viceversa, proporcionando una comunicación bidireccional. Además, puede conectar un puerto y un rol con distinta interfaz, siempre y cuando una de ellas sea un subconjunto de la otra, pero esto obliga a que el attachment se diseñe para una combinación de interfaces específica. Los elementos arquitectónicos a conectar pueden encontrarse en la misma máquina o en distintas máquinas, en cuyo caso la comunicación es distribuida. Además, éstos pueden reconfigurarse dinámicamente, por lo que los attachments deben diseñarse de forma que puedan establecerse nuevas conexiones en tiempo de ejecución.

Por tanto, un attachment debe crearse de forma dinámica para una combinación de interfaces (con servicios comunes), definida por los puertos que conecta. La implementación se ha realizado a través de la herencia, de forma similar a los puertos de entrada. Por una parte, el comportamiento general del attachment se ha definido en una serie de clases genéricas. Por otra parte, el comportamiento específico, dependiente de la interfaz cuyas comunicaciones debe redirigir, es generado en tiempo de compilación a la vez que se generan las interfaces .NET correspondientes a las interfaces PRISMA. Para evitar la generación de un attachment por cada posible combinación de interfaces que pueda conectar, se generan fragmentos de código específicos para cada interfaz. Posteriormente, en tiempo de ejecución, el attachment se construye combinando los fragmentos correspondientes a las interfaces que conecta. De esta forma, por cada interfaz generada, también se genera el código necesario para construir dinámicamente el attachment correspondiente, cuyo comportamiento es heredado de las clases genéricas.

Un attachment tiene dos comportamientos claramente diferenciados: cliente y servidor. El comportamiento cliente consiste en escuchar periódicamente las peticiones depositadas en el puerto de salida que está escuchando, y es por ello que requiere un hilo de ejecución propio. El comportamiento servidor consiste en reenviar las peticiones recogidas al puerto de entrada. Como los elementos arquitectónicos que conecta el attachment pueden estar

en máquinas distintas, se ha subdividido al attachment en dos entidades, cada una de las cuales representa a un participante de la comunicación (componente o conector) y que se ubican en la misma máquina del que representan. Estas entidades son instancias de la clase *Attachment*.

Cada una de estas entidades podrá actuar como cliente o como servidora dependiendo de la dirección en que fluya la comunicación. La que actúe como cliente deberá, tras recoger una petición, reenviarla a la otra entidad, que actuará como servidora. La servidora deberá publicarse a través de Remoting para poder ser accedida de forma remota, por lo que también heredará de *MarshalByRefObject*. Para separar ambos comportamientos se ha optado por implementarlos en clases separadas: *AttachmentClientBase* para el comportamiento cliente, y *AttachmentServerBase* para el comportamiento servidor. Estas clases sólo definen el comportamiento genérico, que es heredado por los attachments específicos generados para cada interfaz en tiempo de compilación, como se ha comentado anteriormente.

Por tanto, un attachment PRISMA se corresponde con una pareja de instancias de la clase *Attachment*, una asociada al componente y otra asociada al conector, cada una de las cuales está formada por la agregación de las clases que implementan el comportamiento cliente y el comportamiento servidor. En la Figura 35 se muestra el esquema general de funcionamiento de los attachments y su composición interna. En cursiva se muestran los nombres de las clases que definen el comportamiento de cada módulo. El esquema muestra el attachment PRISMA definido para conectar dos puertos cuyas interfaces publicadas son "InterfazX" e "InterfazY". Los módulos "Interface{X|Y}Client" e "Interface{X|Y}Server" heredan respectivamente de *AttachmentClientBase* y de *AttachmentServerBase* y son generados en tiempo de compilación para incorporar el comportamiento específico de los attachments.

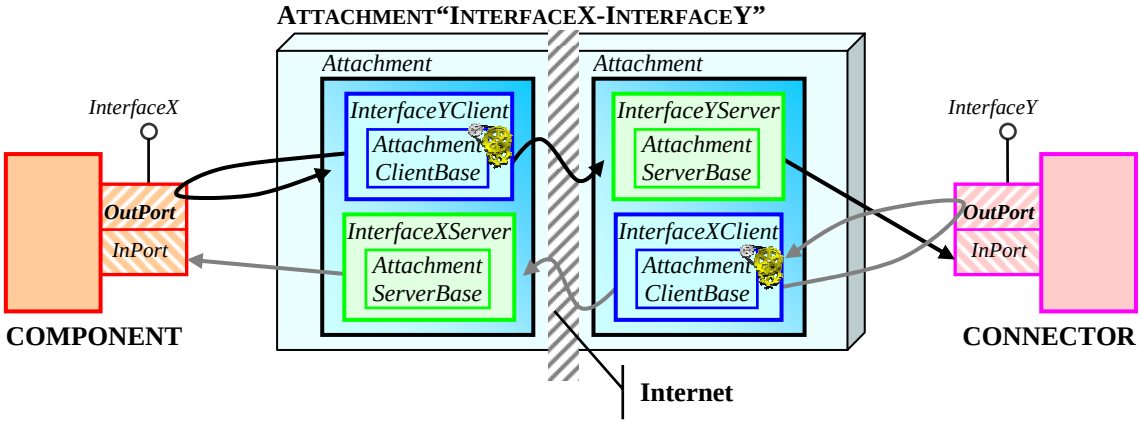

Figura 35 - Esquema de funcionamiento de los Attachments

La comunicación entre un componente y un conector se realiza creando una pareja de attachments, cargando los tipos (Client y Server) correspondientes a la interfaz de los puertos que se desea conectar, instanciándolos e

iniciando su ejecución mediante la llamada a *AttachmentStart()*, como se verá más adelante. Los servicios que el attachment acepta (cliente) y reenvía (servidor) son el resultado de la intersección de los servicios definidos en las interfaces de los puertos. Por ejemplo, si en "InterfaceX" se define un servicio *ChangeName* no definido en "InterfaceY", y se deja en el puerto de salida una petición de este servicio, el módulo "InterfaceYClient" no recogerá dicha petición pues no está contemplada en la interfaz que implementa, y por tanto tampoco podrá solicitarse en el destino dicho servicio. De esta forma, este diseño resuelve el problema de conectar interfaces compatibles sin tener que comparar entre sí los métodos definidos en las interfaces.

Sin embargo, el diseño puede ser optimizado para crear los attachments de forma totalmente dinámica, sin necesidad de generar código, utilizando la reflexión y los delegados. El inconveniente es que los tipos de delegados necesarios deben buscarse en los ensamblados. Por otra parte, el diseño también puede simplificarse si se desea que la comunicación sea unidireccional, eliminando para ello el comportamiento servidor del extremo que actúa exclusivamente como cliente y el comportamiento cliente del extremo que actúa solamente como servidor.

Una vez visto el modelo de ejecución de los attachments, a continuación se describirán los detalles de implementación. En la Figura 36 se muestra el diseño de clases para dar soporte a los attachments. Los attachments se almacenan en una colección del tipo *AttachmentsCollection,* que proporciona los métodos necesarios para añadir y eliminar attachments, y es gestionada por el *middleware*. Como ya se ha comentado, un attachment PRISMA se corresponde con dos instancias de la clase *Attachment*, cada una de las cuales contiene información sobre el componente o conector al cual representan, además de la información necesaria sobre su pareja para poder establecer la conexión remota. En caso de que el otro extremo de la conexión se mueva a otra máquina, el attachment deberá volver a conectarse con su pareja, y para ello necesitará el nombre de su pareja, la URL donde se encuentra y el tipo del objeto servidor que es accesible remotamente.

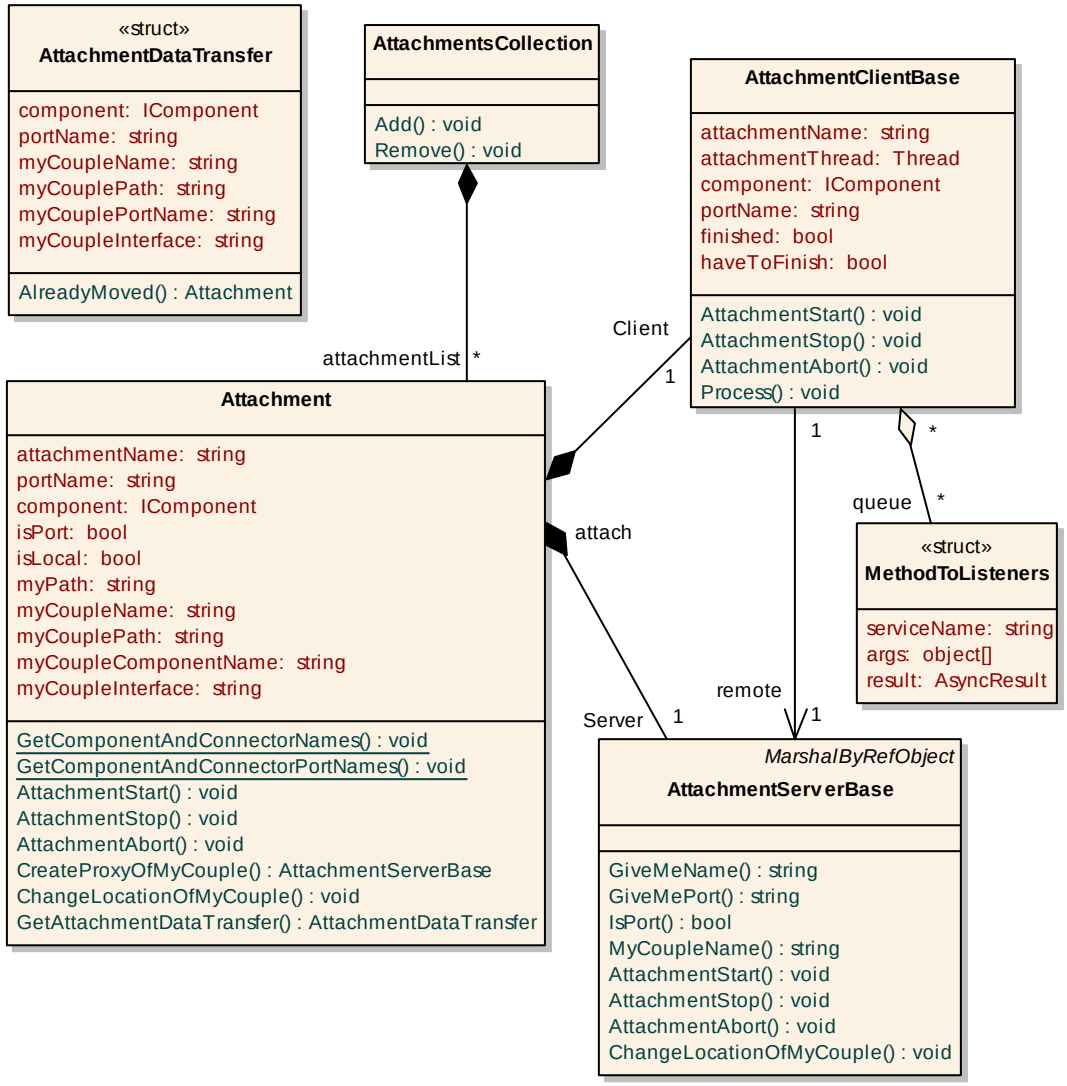

Figura 36 - Clases asociadas al Attachment

El nombre identificativo del attachment debe construirse de tal manera que no provoque ambigüedad en caso de haber más de un attachment conectado al mismo componente y conector. Una alternativa posible es que el nombre esté formado por un identificador numérico, pero el problema es que cada *middleware* generará uno propio, que tal vez coincida con el generado por otro *middleware*. Por esta razón, se ha optado por un mecanismo de identificación más sencillo y global: utilizar una cadena formada por el nombre identificativo del componente y conector que conecta el attachment, además de los puertos que conecta. De esta forma, el attachment queda identificado de forma unívoca por la conexión que establece. Sin embargo, el principal inconveniente de este mecanismo de identificación es que el nombre puede ser largo y por tanto la búsqueda de un determinado attachment en una lista puede ser más costosa que el uso de un identificador numérico.

La clase *Attachment* (ver Figura 36) define los siguientes atributos y servicios:

- **attachmentName**, **portName**, **myPath, component**: atributos básicos de un attachment, como su nombre identificativo, el nombre del puerto al cual escucha (comportamiento cliente) y al cual reenvía peticiones recibidas (comportamiento servidor), la URL donde se encuentra, y la referencia al componente al cual el *attachmentServer* actúa como proxy. A través de esta referencia el attachment podrá invocar los servicios del puerto de entrada y recoger las peticiones del puerto de salida.
- **isPort**, **isLocal**: flags que permiten conocer si la instancia actual representa a un componente (*isPort=true*) o a un conector (*isPort=false*), o si el par de attachments está en la misma máquina (*isLocal=true*) o en máquinas distintas (*isLocal=false*).
- **myCoupleName**, **myCouplePath**, **myCoupleComponentName**, **myCoupleInterface**: atributos que permiten conocer los datos del otro miembro de la pareja del attachment actual. Son utilizados para establecer la comunicación entre el Cliente y el Servidor.
- **Client**, **Server**: atributos que contienen las referencias a los objetos agregados *AttachmentClientBase* y *AttachmentServerBase*, respectivamente.
- **GetComponentAndConnector[Names|PortNames]()**: métodos estáticos que permiten obtener, a partir del nombre identificativo de un attachment, el nombre del componente y conector o el puerto y rol que une.
- **Attachment[Start|Stop|Abort]()**: servicios que inician, paran o detienen la ejecución de un attachment. El inicio de un attachment implica su registro en los puertos de entrada y de salida, y la interconexión entre los miembros de la pareja de attachments. De la misma forma, la parada implica la desuscripción de los respectivos puertos. A su vez, la invocación de estos servicios es propagada al objeto agregado *Client* para el inicio, parada o detención del hilo de ejecución responsable de escuchar las peticiones depositadas en el puerto de salida.
- **CreateProxyOfMyCouple()**: servicio para crear un proxy hacia el objeto *server* del otro attachment. Se utiliza para poder acceder al otro attachment cuando se encuentra en una máquina distinta.
- **ChangeLocationOfMyCouple()**: servicio que actualiza la ubicación donde se encuentra la otra pareja del attachment, con el objetivo de que vuelva a establecerse la comunicación entre los dos miembros de la pareja (el cliente se conecte al servidor remoto y el cliente remoto se conecte al servidor local).
- **GetAttachmentDataTransfer()**: servicio de creación de un objeto serializable (de tipo *AttachmentDataTransfer*) con la información relevante del attachment para que cuando éste vaya a ser movido a otra máquina pueda reconstruirse de nuevo manteniendo el estado.

La invocación del método *AlreadyMoved()* de este objeto crea de nuevo el attachment con los datos transferidos.

La clase *AttachmentClientBase* define el comportamiento cliente del attachment. Tiene los atributos necesarios para conectarse al puerto del componente (o conector) del cual recoge peticiones, y un atributo *queue* que es una referencia a la cola exclusiva proporcionada por el puerto de salida, en la que se van depositando las peticiones dirigidas al attachment. Para ello, el cliente tiene un hilo de ejecución propio que cada cierto tiempo comprueba el estado de dicha cola.

Esta alternativa proporciona la máxima independencia del attachment respecto al puerto de salida, ya que éste en ningún momento interactúa directamente con el attachment, tan sólo va depositando mensajes para que sean recogidos. Sin embargo, como desventaja dicho hilo de ejecución ocupa (pequeños) ciclos de CPU para comprobar si hay peticiones en la cola, que en caso de no haberlas, se convierte en una operación innecesaria. Otra posible alternativa podría ser que dicho hilo estuviese en estado suspendido hasta que el puerto de entrada dejase una petición en su cola y lanzase un evento que lo despertase. De esta forma, el hilo del cliente únicamente estaría activo cuando realmente tuviese peticiones que procesar. No obstante, el puerto de salida necesitaría acceder al hilo de ejecución del attachment para despertarlo, lo que conlleva una pérdida de independencia del puerto.

Los flags *finished* y *haveToFinish* son indicadores del estado de ejecución de dicho hilo y que permiten detenerlo cuando sea invocado el servicio *AttachmentStop*. El atributo *remote* contiene un proxy hacia el *Server*, que puede ser local o remoto, al cual se le reenvían las peticiones recibidas. El servicio *Process()* es invocado cada vez que se detecta una nueva petición y debe ser sobrecargado por el *AttachmentClient* específico generado para la interfaz concreta.

A continuación se muestra el código generado para el attachment específico de la interfaz *ICreditCardTransactions*, comportamiento cliente. El nombre del tipo generado es: *AttachmentICreditCardTransactionsClient*

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace CuentaBancaria {

   // Clase generada automáticamente al generarse el código de la interfaz
   // "ICreditCardTransactions"
   [Serializable]
   public class AttachmentICreditCardTransactionsClient: AttachmentClientBase {

      // Constructor - Llama al padre
      public AttachmentICreditCardTransactionsClient
              (IComponent component, string portName, string attachmentName)
           : base(component, portName, attachmentName) {}

      // Procesa una petición de servicio
```

```
      public override void Process(MethodToListeners service) {
         try {
            switch(service.serviceName) {
               case "Withdrawal": {
                  decimal arg0 =(decimal)service.Args[0];
                  decimal arg1 =(decimal)service.Args[1];
                  AsyncResult result = ((ICreditCardTransactions) remote).
                     Withdrawal(arg0, ref arg1);
                  service.Result.SetDerivedParameters(result);
                  break;
               }
               case "Balance": { /*[Omitido]*/ }
               case "ChangeAddress": { /*[Omitido]*/ }
               case "Transfer": { /*[Omitido]*/ }
               default:
                  break;
            }
         }
      }
   }
}
```

Como puede observarse, la mayor parte del código generado corresponde al cuerpo del método *Process*, en el que existe un "case" para cada servicio ofrecido por la interfaz. Cuando en la cola se reciba una petición de servicio, el cliente comprobará si el nombre del método pertenece a uno de los servicios de la interfaz, y en caso afirmativo, se invocará al servicio correspondiente del Server. Todos los servicios invocados en el AttachmentServer devuelven un proxy al objeto *AsyncResult* almacenado en la máquina destino, por lo que debe establecerse una relación entre el objeto *AsyncResult* local y el remoto, para que cuando el remoto obtenga los resultados, el local sea avisado convenientemente. Esto se realiza invocando el método *setDerivedParameters* del objeto *AsyncResult* local, almacenado en la cola del puerto de salida, y consultado por el aspecto que ha invocado el servicio externo.

Finalmente, la clase *AttachmentServerBase* es la que define el comportamiento servidor del attachment. Por una parte, actúa como objeto accesible remotamente que publica los servicios de una determinada interfaz y redirige las peticiones al componente/conector del cual actúa como proxy. Por otra parte, también actúa como proxy de la instancia *Attachment* a la cual pertenece. Esto se ha realizado para evitar publicar también la instancia de *Attachment*. Ésta debe ser accedida remotamente para la parada de los attachments o para actualizarla en caso de que el otro miembro de la pareja se haya movido a una máquina distinta.

A continuación se muestra el código generado para el attachment específico de la interfaz *ICreditCardTransactions*, comportamiento servidor. El nombre del tipo generado es: *AttachmentICreditCardTransactionsServer*.

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;
```

```
namespace CuentaBancaria {

   // Clase generada automáticamente al generarse el código de la interfaz
   // "ICreditCardTransactions"
   [Serializable]
   public class AttachmentICreditCardTransactionsServer: AttachmentServerBase,
                  ICreditCardTransactions {
      private ICreditCardTransactions InPortOfComponent;

      // Constructor - Llama al padre
      public AttachmentICreditCardTransactionsServer(Attachment attach):
                base (attach) {

         InPortOfComponent =
              (ICreditCardTransactions) Component.InPorts[attach.PortName];
         if (InPortOfComponent == null) {
            throw new Exception("ICreditCardTransactions's InPort does not
               exists in " + Component.componentName + " component.");
         }
      }

      /* REDIRECCIÓN DE LOS SERVICIOS HACIA EL COMPONENTE DESTINO */

      public PRISMA.AsyncResult Withdrawal(decimal quantity,ref decimal money){
         try {
            return InPortOfComponent.Withdrawal(quantity, ref money);
         } catch (Exception e) {
            // Propagamos la excepción al cliente del servicio
            AsyncResult result = new AsyncResult(1);
            result.ThrowExceptionToClient(e);
            return result;
         }
      }

      public PRISMA.AsyncResult Balance(ref decimal money){/*[Omitido]*/}
      public PRISMA.AsyncResult ChangeAddress(string newAdd){/*[Omitido]*/}
      public PRISMA.AsyncResult Transfer(decimal quantity, ref decimal money) {
         /*[Omitido]*/ }
   }
}
```

Como puede observarse, el código generado también es mínimo. En el constructor, por motivos de eficiencia, se almacena en una variable de clase una referencia al puerto de entrada al cual el Server debe reenviar las peticiones recibidas. Por cada servicio definido en la interfaz, se define un método con la misma signatura cuya funcionalidad es reenviar la misma petición al puerto de entrada del componente o conector, y capturar las excepciones para propagarlas al cliente del servicio. Las excepciones son retransmitidas a través del objeto *AsyncResult*, mediante la invocación del método *ThrowExceptionToClient*().

4.2.5.3 Bindings

En PRISMA, un binding es la entidad encargada de establecer la comunicación entre un componente/conector y el sistema que lo encapsula. Asocia un puerto de un sistema con un puerto de un componente o conector, actuando como un redirector de peticiones, de forma transparente para los componentes externos al sistema. Cuando un puerto de un sistema recibe

una petición de servicio, y éste tiene un binding asociado, la petición es reenviada al componente interno, que puede estar en otra máquina distinta a la del sistema. Por el contrario, cuando un componente/conector interno desea comunicarse con un componente externo, tras dejar la petición en su puerto de salida, el binding recoge dicha petición y la reenvía al puerto de salida del sistema.

Su diseño se ha realizado de forma similar a los attachments. Un binding está formado por dos partes, cada una de las cuales reside en la misma máquina que la entidad a la cual representa. La diferencia radica en que la parte que representa al sistema se ha diseñado de forma específica para que sea común para todos los bindings de dicho sistema. En la Figura 37 se muestra el esquema de funcionamiento de un binding PRISMA.

La parte que reside en el lado del componente/conector, a la que se ha denominado *ComponentBinding*, conserva el mismo diseño de los attachments: está formada por un objeto que implementa el comportamiento cliente (*ComponentBindingClient*) y otro objeto con el comportamiento servidor (*ComponentBindingServer*). El objeto cliente se encarga de recibir las peticiones de servicio del puerto de salida del componente y redirigirlas hacia el otro extremo de la comunicación, la parte residente en el lado del sistema. El objeto servidor se encarga de recibir las peticiones enviadas por la parte del binding residente en el sistema y redirigirlas al puerto de entrada del componente.

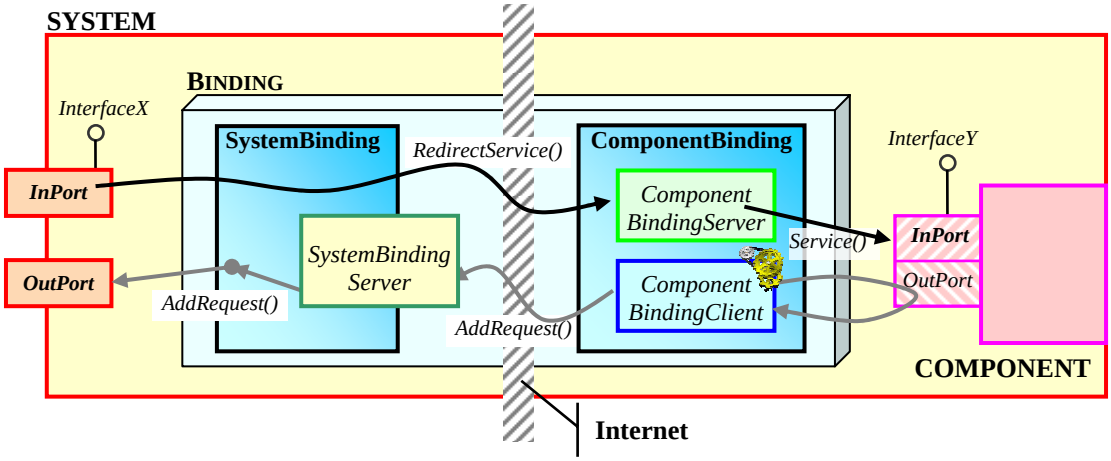

Figura 37 - Esquema de funcionamiento de un Binding

La parte que reside en el lado del sistema, a la que se ha denominado *SystemBinding*, realiza dos tareas:

- Realiza las funciones complementarias al *ComponentBinding*. Por una parte, recoge todas las peticiones de servicio salientes de los componentes internos, enviadas a través de los *ComponentBindings,* y las reenvía a los puertos de salida correspondientes. Por otra parte, inicializa adecuadamente los puertos de entrada para que redirijan las peticiones de servicio entrantes hacia los componentes internos.

- Gestiona los bindings del sistema, encargándose de la creación y eliminación de bindings. Existe sólo una instancia de *SystemBinding* por cada sistema, y es la encargada de crear los respectivos *ComponentBindings* por cada binding definido. También es la encargada de actualizar las referencias en caso de movilidad. Cuando detecta que existen *ComponentBindings* remotos, publica una parte de sí misma (*SystemBindingServer*) para ser accedida remotamente.

A diferencia de los attachments, los cuales requieren dos hilos de ejecución por cada módulo cliente (uno en cada máquina) para realizar la escucha de los puertos de salida, los bindings se han implementado de forma diferente para mejorar el rendimiento. La parte del binding residente en el sistema (*SystemBinding*) no requiere escuchar de forma activa los puertos de entrada y salida del sistema, pues no existen colas de espera. Las peticiones de servicio recibidas por el puerto de entrada pueden ser redireccionadas inmediatamente al *ComponentBinding*, y lo mismo ocurre con las peticiones enviadas por éste al *SystemBinding*: sólo hay que reenviarlas al puerto de salida correspondiente. Por esta razón, no ha sido necesario un hilo de ejecución propio para el *SystemBinding*, aunque los *ComponentBindings* sí que lo requieren para la escucha periódica de las peticiones generadas por el puerto de salida del componente interno.

Habrá tantos *ComponentBindings* como bindings haya definidos en el sistema, pero sólo un único *SystemBinding* para todos ellos. Esto se ha realizado de esta forma con la finalidad de encapsular en una única clase toda la funcionalidad necesaria para la gestión de los bindings definidos en un sistema, y para evitar la publicación de un *SystemBindingServer* por cada binding definido. De esta forma, todos los bindings remotos de un mismo sistema confluyen a un único objeto, que es el que gestiona la concurrencia para acceder a los puertos del sistema.

El modelo de ejecución de los bindings es semejante al de los attachments. Por una parte, cuando el *ComponentBinding* detecta una petición de servicio en el puerto de salida del componente al cual está escuchando, invoca el método *AddRequest()* del *SystemBinding* proporcionándole el nombre del *ComponentBinding*, el servicio a ejecutar y sus parámetros. Dicho método, tras comprobar que ha sido invocado por un *componentBinding* válido, obtiene el puerto del sistema al cual redirigir la petición e invoca al método *AddRequest()* del puerto de salida correspondiente del sistema, pasándole los parámetros recibidos desde el *componentBinding*.

Por otra parte, la redirección de las peticiones de servicio entrantes hacia los *ComponentBindings* se ha realizado a través de un atributo de los puertos de entrada: *ComponentBinding*. Cuando se asocia un binding a un puerto, el objeto *SystemBinding* inicializa su atributo *ComponentBinding* con una referencia al *ComponentBindingServer* (que puede ser remoto o local). En tiempo de ejecución, cuando llega una nueva petición a un sistema, el puerto de entrada crea un delegado hacia el método *RedirectService* del *ComponentBindingServer* y lo almacena en la cola del sistema, con la

finalidad de preservar el orden de llegada de las peticiones. Posteriormente, al ser ejecutado dicho delegado por el sistema, la petición de servicio será reenviada al puerto de entrada del componente asociado. Este diseño presenta el inconveniente de que ha precisado modificar el puerto de entrada del sistema para que también se almacenase el binding asociado, resultando en una pérdida de independencia por parte de los puertos. Sin embargo, esto ha sido necesario porque el único punto donde pueden interceptarse las peticiones de servicio para redirigirlas hacia los bindings es en los puertos de entrada, ya que tras ser insertadas en la cola del sistema se pierde la información sobre el puerto de la cual procedían. Además, tampoco pueden redirigirse directamente, sin pasar por la cola del sistema, las peticiones recibidas en los puertos de entrada hacia los *ComponentBindings*, ya que en ese caso se perdería el orden de los servicios recibidos.

Los bindings, a diferencia de los attachments, son creados dinámicamente sin necesidad de generar código, gracias a la reflexión y al uso de los delegados proporcionados por .NET. Cuando se crea un nuevo binding, el *SystemBinding* indica al middleware que cree un nuevo *ComponentBinding* en la ubicación del componente remoto, indicándole entre otra información, el nombre del sistema con el cual se comunicará y el puerto y nombre del componente al cual debe reenviar peticiones. El constructor del *ComponentBinding* crea a su vez las instancias *ComponentBindingClient* y *ComponentBindingServer*, proporcionándoles la interfaz del puerto de salida del componente y del puerto de entrada del sistema, respectivamente. A partir de la interfaz proporcionada, *ComponentBindingClient* obtiene dicho tipo a través del middleware y por reflexión obtiene los métodos que la forman, añadiéndolos a una lista dinámica. En tiempo de ejecución, los servicios recogidos del puerto de salida son comparados con los de dicha lista, y en caso de coincidir, son reenviados al *SystemBinding*. Por su parte, y de forma similar, *ComponentBindingServer* obtiene los métodos de la interfaz a través de la reflexión, pero en lugar de almacenar únicamente los nombres de los servicios, crea también un delegado hacia el método destino. En tiempo de ejecución, cuando el método *RedirectService* es invocado, al que se le proporciona el nombre del método a ejecutar, se obtiene el delegado correspondiente y se ejecuta con los parámetros proporcionados. Otra posible estrategia para la invocación del servicio solicitado podría ser, en lugar de utilizar delegados, utilizar el método *Invoke()* que se proporciona por la reflexión. Sin embargo, y como ya se discutió en el apartado 4.2.4.3, el uso de delegados es más eficiente que la invocación de métodos por reflexión ([Gun04]).

Por otra parte, los bindings también implementan mecanismos para el inicio o la parada segura, a través de los métodos *Start* y *Stop* implementados tanto en el *SystemBinding* como en los *ComponentBindings*. En el *ComponentBinding*, la invocación del método *Start* supone el inicio del hilo de ejecución del *ComponentBindingClient*, mientras que el *Stop* supone la parada de dicho hilo, con lo que no se recogen nuevas peticiones. Sin

embargo, debido a que *SystemBinding* y *ComponentBindingServer* no disponen de un hilo de ejecución propio, el inicio y la parada se indica a través de un flag interno, *stopped*, que impide que se procesen y reenvíen peticiones de servicio. Se contemplan dos tipos de paradas, la parada del sistema o la parada de los componentes internos. En caso de que el sistema deba pararse (p. ej. porque va a ser movido), deben detenerse también todos los bindings asociados, lo que implica la parada de todos los *ComponentBindings*. Esto es realizado por el método *Stop()* del *SystemBinding*, al igual que el método *Start()* se encarga de reanudar la ejecución de los *ComponentBindings* asociados. En el caso de la parada de un componente interno, también deben ser parados los bindings que tenga asociados, al igual que los attachments. Esta tarea es realizada por el middleware a través de la invocación de los métodos *Start()* y *Stop*() de los *ComponentBindings*. El inicio o la parada puede ser invocada de forma local o remota a través del *ComponentBindingServer*, que propaga la petición al *ComponentBinding* y éste a su vez también detiene el módulo cliente.

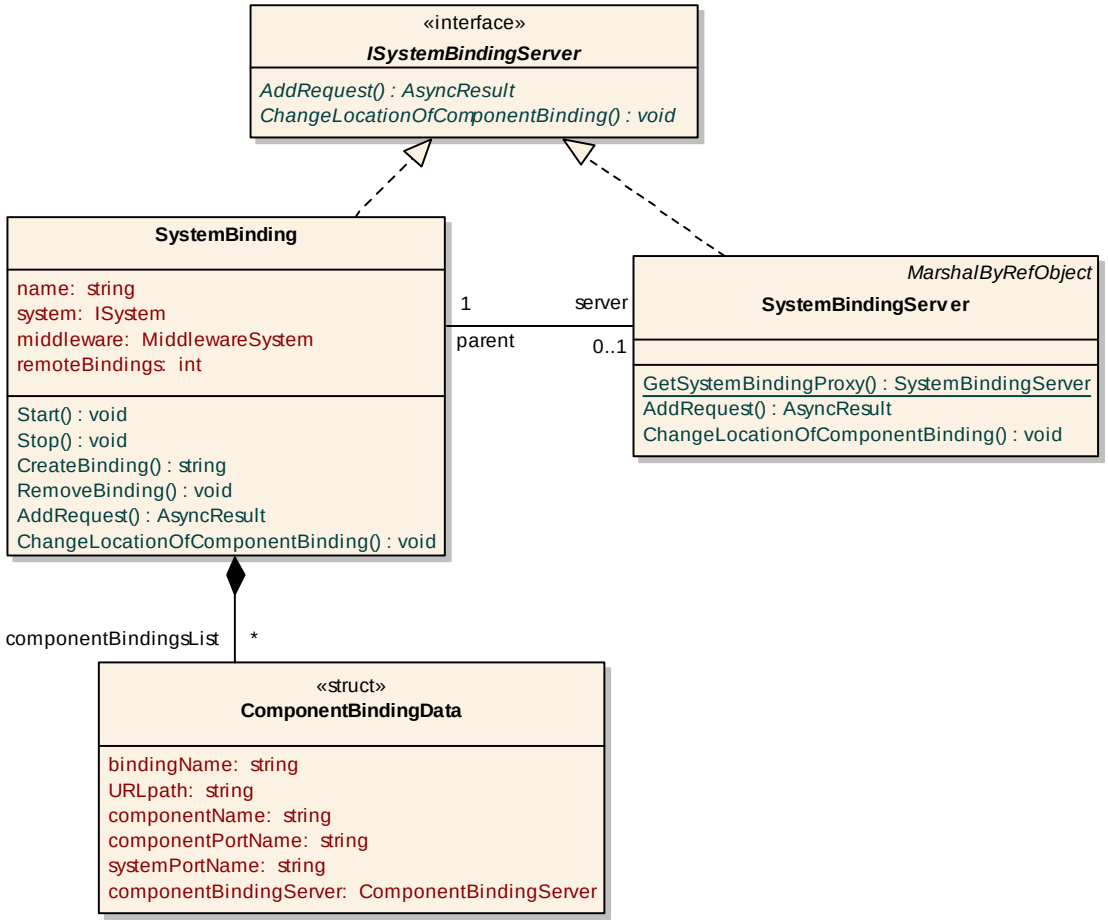

Figura 38 - Clases SystemBinding y SystemBindingServer

Una vez visto el diseño de los bindings, a continuación se describirán las características de las clases citadas anteriormente. La primera de ellas es la clase *SystemBinding* (ver Figura 38), cuyos atributos y servicios son:

- **name**: atributo que almacena el nombre identificativo de la instancia. Se construye a partir del nombre del sistema al cual pertenece, y se utiliza para crear el nombre de los *ComponentBindings*.
- **system**, **middleware**: atributo que guarda las referencias hacia el sistema al cual pertenece el gestor de bindings y hacia el middleware.
- **remoteBindings**: atributo que indica el número de bindings remotos que están utilizando el objeto *SystemBindingServer*. Este contador permite conocer cuándo debe publicarse o no dicho proxy.
- **server**: atributo para guardar la referencia hacia el *SystemBindingServer* creado.
- **componentBindingsList**: atributo que contiene una lista con la información sobre los distintos *ComponentBindings* creados. Permite conocer a qué puerto debe redireccionarse cada petición de servicio recibida desde los *ComponentBindings*, o volver a conectarse a ellos en caso de movimiento del sistema a otra máquina. Dicha información se almacena en tuplas *ComponentBindingData*, cuyos campos contienen el nombre identificativo del *ComponentBinding*, la URL donde se encuentra, el nombre del puerto y del componente al cual representa, el nombre del puerto del sistema, y una referencia a la parte servidora, que en caso de ser remota será un proxy.
- **Start()**, **Stop()**: servicios para iniciar o detener la ejecución de los bindings, respectivamente.
- **CreateBinding()**, **RemoveBinding()**: servicios para crear o eliminar un binding de forma dinámica, de la forma que se ha descrito anteriormente.
- **AddRequest()**: servicio que añade la petición de un servicio al puerto de salida del sistema, dado el nombre del *ComponentBinding* que invoca el método, el nombre del servicio requerido y sus argumentos.
- **ChangeLocationOfComponentBinding()**: servicio que actualiza con el nuevo valor el nodo correspondiente de la lista de *ComponentBindings*, dada una referencia a un objeto *ComponentBindingServer*. Se utiliza para reflejar el cambio de ubicación de un *ComponentBinding*, normalmente a causa de que su componente asociado se ha movido a otra máquina.

Por otra parte, los atributos y servicios de la clase *ComponentBinding* (Figura 39) se describen seguidamente:

- **name**: atributo que define el nombre del binding, para que pueda ser identificado de forma unívoca por el *middleware*. El nombre es creado a través del método estático *CreateName()*.
- **portName**, **myPath**, **componentRef**: atributos que contienen la información relativa al componente al que representa, como el nombre del puerto al que se debe escuchar y reenviar peticiones, la

URL donde se encuentra la instancia de *ComponentBinding*, y una referencia para poder acceder al componente.

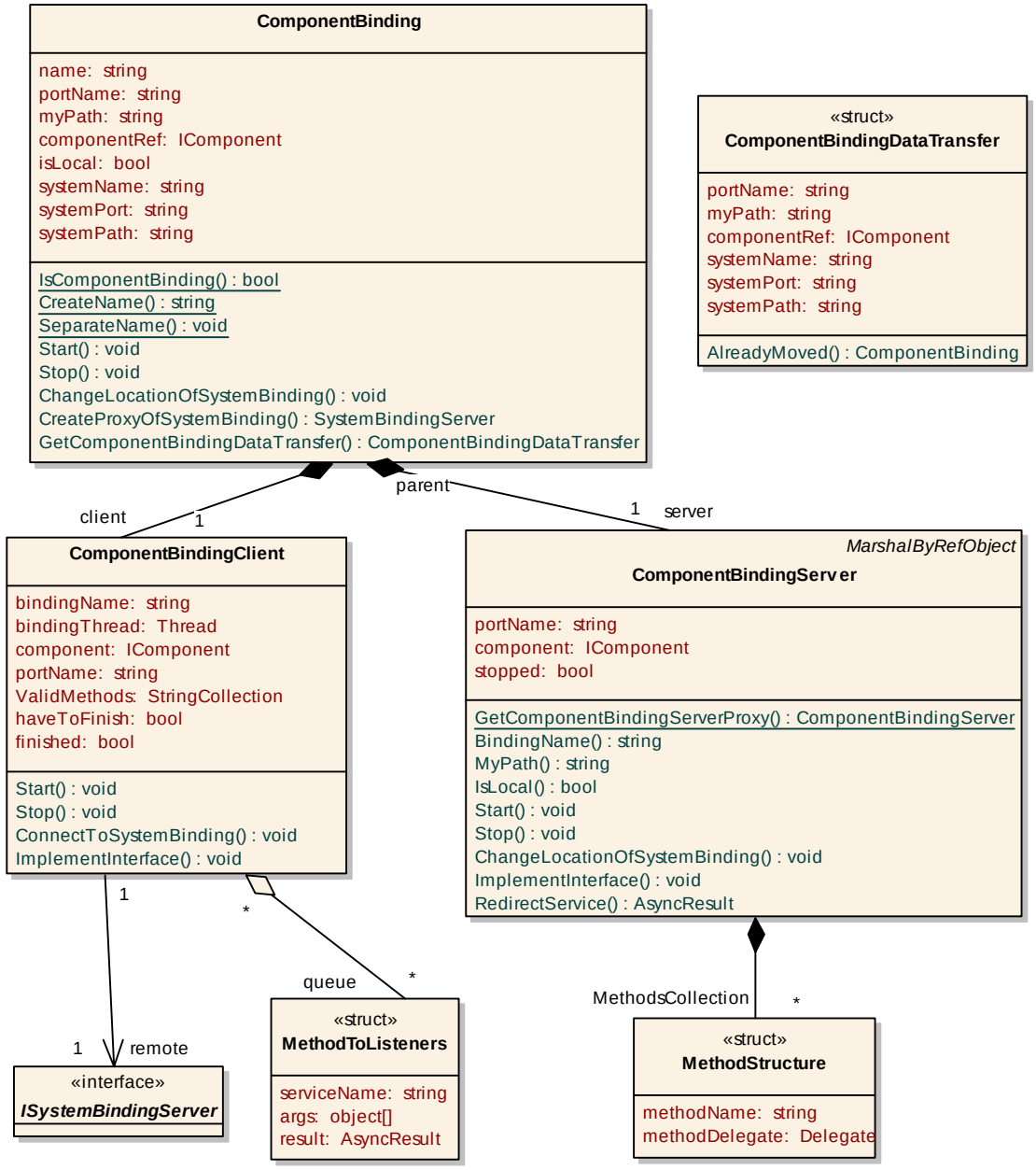

Figura 39 - Clases asociadas al ComponentBinding

- **isLocal, systemName, systemPort, systemPath**: atributos que definen información relativa al sistema al cual este *componentBinding* pertenece, como el nombre, el puerto y la URL donde se encuentra. El atributo *isLocal* indica si este *componentBinding* es local al *systemBinding*, por lo que no requiere la publicación del objeto *componentBindingServer* como objeto remoto.
- **Client, Server**: atributos que contienen las referencias al módulo servidor (*ComponentBindingServer*) y al módulo cliente (*ComponentBindingClient*).

- **IsComponentBinding()**: método estático que indica si es un *ComponentBinding* o no (en cuyo caso será un attachment), dado un nombre de un *listener* de un puerto de entrada o de salida.
- **CreateName()**, **SeparateName()**: métodos estáticos que definen cómo se construye un nombre identificativo de un *ComponentBinding*. Dicho nombre se construye mediante la concatenación del nombre y puerto del sistema y el nombre y puerto del componente/conector entre los cuales establece la comunicación, de forma similar a los attachments. El método *SeparateName* separa las partes que componen dicha cadena.
- **Start()**, **Stop()**: métodos para iniciar o parar la ejecución del *ComponentBinding* de forma segura, como se ha comentado anteriormente.
- **ChangeLocationOfSystemBinding()**, **CreateProxyOfSystemBinding()**: métodos para actualizar las referencias hacia el *SystemBinding*, o crear el proxy correspondiente para acceder remotamente.
- **GetComponentBindingDataTransfer()**: método que genera una estructura con el estado del *ComponentBinding*, cuyo tipo es *ComponentBindingDataTransfer*, de igual forma que los attachments. Esta estructura es utilizada para mover el *ComponentBinding* a otra máquina, y una vez allí, a través de la invocación del método *AlreadyMoved*() se vuelve a generar el *ComponentBinding*.

La clase agregada *ComponentBindingClient* es muy similar al *AttachmentClient*, pues tiene un hilo de ejecución propio (*bindingThread*) encargado de escuchar las peticiones de servicio del puerto de salida del componente, cuyo nombre se almacena en *portName*. La parada de dicho hilo se controla a través de los atributos privados *haveToFinish* y *isFinished*. El primero es activado cuando se invoca el servicio *Stop()*, y el segundo sólo es activado cuando finalmente el hilo se ha parado de forma segura. Las peticiones generadas por el puerto de salida son almacenadas en una cola, cuya referencia es almacenada en el atributo *queue* al registrarse el *ComponentBindingClient* a dicho puerto. Los servicios válidos que se aceptarán y serán reenviados hacia el *SystemBinding* (cuya referencia es almacenada en el atributo *remote*) son los definidos en la lista *ValidMethods*, obtenidos dinámicamente a partir de la interfaz del puerto de salida del componente (método *ImplementInterface*).

Por otra parte, la clase agregada *ComponentBindingServer* acepta las peticiones enviadas desde el *SystemBinding* y las redirecciona al componente al cual está asociada. Hereda de *MarshalByRefObject* para que pueda ser accedida remotamente si se encuentra en una máquina distinta a la del *SystemBinding*. Las peticiones de servicio son recibidas mediante la invocación remota del método *RedirectService*, proporcionando el nombre del servicio a ejecutar y un vector con sus parámetros. Tras obtener de la lista

MethodsCollection el delegado asociado al nombre del servicio, éste es invocado con los parámetros proporcionados. Esta lista de delegados es construida por el método *ImplementInterface* a partir de la interfaz del puerto de entrada del componente, como se ha comentado anteriormente.

4.2.5.4 Definición de Sistemas en .NET

Una vez visto el funcionamiento de los attachments y de los bindings, y los mecanismos de comunicación que proporcionan, en esta sección se muestra cómo éstos son definidos en un sistema para establecer las comunicaciones entre los distintos elementos arquitectónicos que lo componen. De igual forma, también se presentará cómo los componentes y conectores son agregados al sistema.

Los sistemas también utilizan la herencia para implementar los tipos PRISMA, de la misma forma que los componentes. En la clase *SystemBase* se define el comportamiento genérico de los sistemas, mientras que el comportamiento específico es generado en tiempo de compilación para cada tipo PRISMA definido. Para ello, se tomará como ejemplo la especificación del sistema *BankSystem*, mostrada en el apartado 3.4.1.4:

```
System_type BankSystem
  Ports
    CreditCard_port : ICreditCardTransactions;
  End_Ports

  Variables
    Var_ATM : ATM;
    Var_Account : Account;
    Var_ATMAccount : ATMAccount;
  End_Variables

  Attachments
    Var_ATM.AccountTrans_port ↔ Var_ATMAccount.ATM_role;
    Var_Account.Account_port  ↔ Var_ATMAccount.Account_role;
  End_Attachments;

  Bindings
    Var_ATM.VISACreditCard_port ↔ BankSystem.CreditCard_port;
  End_Bindings;

  Initialize
    New() {
      Var_ATM = new ATM(accountId: decimal, initialLocation: LOC);
      Var_Account=new Account(accountId:decimal,initialLocation:LOC);
      Var_ATMAccount = new ATMAccount(initialLocation: LOC);
    }
  End_Initialize;

  Destruction
    Destroy() {
      Var_ATM.destroy();
      Var_Account.destroy();
      Var_ATMAccount.destroy();
    }
  End_Destruction;
```

```
End_System_type BankSystem;
```

A continuación se muestra el código generado para definir dicho sistema:

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace CuentaBancaria {

   // Definición del Sistema BankSystem
   [Serializable]
   public class BankSystem: SystemBase {

      public BankSystem (string name, MiddlewareSystem middlewareSystem,
            <Atributos_requeridos_por_los_componentes_internos>
            decimal ATM_Id, LOC ATM_initialLocation,
            decimal Account_Id, LOC Account_initialLocation,
            LOC ATMAccount_initialLocation
            <\Atributos_requeridos_por_los_componentes_internos>
               ) : base(name, middlewareSystem) {

      <Definicion_Aspectos>
         // No se han definido en el ejemplo
      <\Definicion_Aspectos>

      <Definicion_Weavings>
         // No se han definido en el ejemplo
      <\Definicion_Weavings>

      <Definicion_Puertos>
         // Creación de PUERTOS de ENTRADA y SALIDA
         InPorts.Add("CreditCardPort", "ICreditCardTransactions");
         OutPorts.Add("CreditCardPort", "ICreditCardTransactions", "");
      <\Definicion_Puertos>

      <Definicion_Componentes>
         // Var_ATM
         AddComponent("Var_ATM", typeof(CuentaBancaria.ATM),
                     ATM_initialLocation, ATM_Id);

         // Var_Account
         AddComponent("Var_Account", typeof(CuentaBancaria.Account),
                     Account_initialLocation, Account_Id);
      <\Definicion_Componentes>

      <Definicion_Conectores>
         // Var_ATMAccount
         AddConnector("Var_ATMAccount", typeof(CuentaBancaria.ATMAccount),
                     ATMAccount_initialLocation);
      <\Definicion_Conectores>

      <Definicion_Attachments>
         AddAttachment("Var_ATM", "AccountTrans_port", "Var_ATMAccount",
                           "ATM_role");

         AddAttachment("Var_Account", "Account_port", "Var_ATMAccount",
                           "Account_role");
      <\Definicion_Attachments>

      <Definicion_Bindings>
         AddBinding("CreditCard_port", "Var_ATM", "VISACreditCard_port");
      <\Definicion_Bindings>
```

```
        }
    }
}
```

Un sistema PRISMA se implementa a través de una clase que hereda su comportamiento de la clase *SystemBase*, y es *Serializable* para poder ser distribuida a otra máquina. De la misma forma que el componente, la estructura de un sistema se define en el cuerpo del constructor y también se encuentra separada en distintas secciones. Algunas de ellas son comunes a los componentes, como la definición de aspectos, weavings y puertos. En cambio, las que caracterizan a un sistema son las que permiten definir componentes, conectores, attachments y bindings. Como se puede observar, la conversión desde la especificación PRISMA a la implementación es casi directa.

Para agregar un componente o conector se invoca el método *AddComponent* o *AddConnector* proporcionando el nombre del componente o conector, el tipo, la localización inicial y los valores de inicialización adicionales requeridos por cada tipo. Puede observarse en el ejemplo que la localización inicial es proporcionada a través del tipo LOC, estructura creada para identificar la localización de un componente distribuido. Contiene un atributo de localización e incorpora métodos para comprobar que dicho valor es una URI bien definida. El *middleware* se encargará de ubicar en la localización especificada a los componentes/conectores creados.

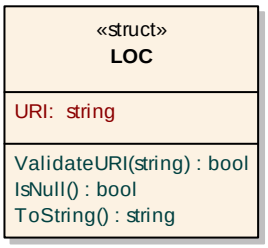

Figura 40 - Estructura LOC

Del mismo modo, la definición de los attachments se realiza especificando el nombre del componente y del puerto, junto al nombre del conector y del rol. La definición de los bindings se realiza especificando el puerto del sistema junto al nombre del componente/conector y del puerto/rol.

4.2.6 MiddlewareSystem

La clase *MiddlewareSystem* implementa la funcionalidad más dependiente de la plataforma de desarrollo, y proporciona distintos servicios a los elementos arquitectónicos PRISMA.

- Gestiona la creación y destrucción de objetos
- Permite establecer la comunicación distribuida, por lo que es el encargado de la creación de los attachments y los bindings.

- Proporciona los servicios de movilidad y de transferencia de componentes, conectores y sistemas entre los diferentes *middleware*.
- Proporciona servicios de localización de los distintos componentes, es decir, es el encargado de realizar las correspondientes búsquedas DNS entre los diferentes *middleware*
- Establece las comunicaciones necesarias con otros *middleware* para propagar las solicitudes de servicio que no puede atender.
- Se encarga de la búsqueda de los diferentes tipos solicitados o requeridos por los elementos arquitectónicos, bien cargando los ensamblados desde la máquina local o solicitándolos a otros *middleware*.
- Implementa un servicio de suscripción para enviar los mensajes de depuración a los suscriptores.

Además, también proporcionará una serie de servicios, de cara a soportar la evolución, para propagar los cambios a los distintos *middleware*, tanto de los elementos arquitectónicos en ejecución, como de los tipos (ensamblados) de los otros *middleware*, aunque queda propuesto como tarea futura.

En la Figura 41 se muestra el diagrama de despliegue del modelo de ejecución PRISMA, soportado por varios *middlewares* ubicados en distintos nodos. Cada uno de los *middlewares* tendrá una instancia en ejecución de la clase *middlewareSystem*.

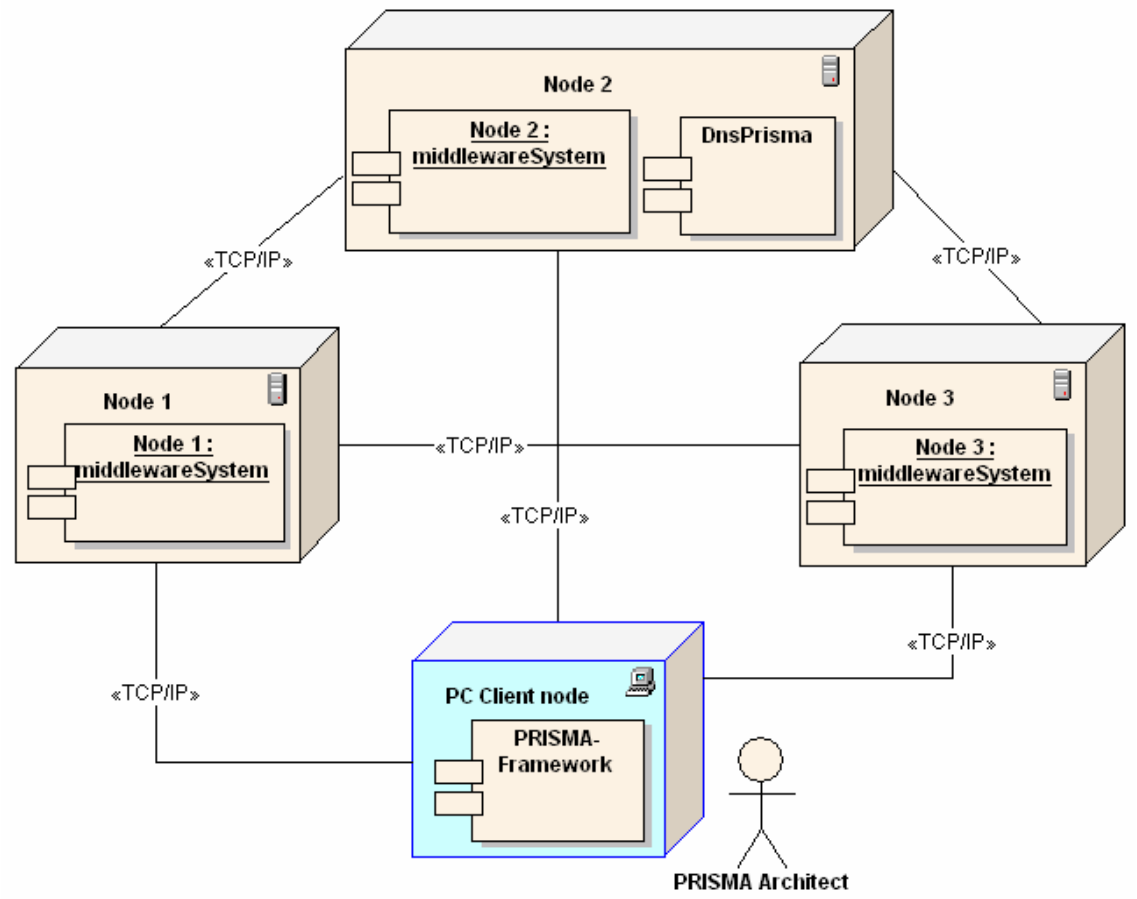

Figura 41 - Diagrama de despliegue del middleware

El nodo *PC Client* contendrá el framework de PRISMA y desde él se podrán controlar de forma centralizada los diferentes *middlewares*. Su cometido es el de administrar los modelos PRISMA en ejecución, cargarlos o detenerlos, así como reconfigurar sus componentes o hacerlos evolucionar, etc. Este nodo también es el que recibe los mensajes de depuración de los diferentes *middlewares*, con lo que puede controlar su estado, a través del mecanismo de suscripción que se comentará a continuación.

La clase *DnsPrisma* modela un DNS centralizado. Esta clase es publicada para su acceso remoto en una ubicación conocida y los distintos *middleware* se conectan a ella para añadir, borrar o consultar ubicaciones de componentes, conectores o sistemas. Sin embargo, en un futuro esta clase se implementará de forma descentralizada, manteniendo su interfaz intacta, de acuerdo con el modelo PRISMA. Cada *middleware* tendrá una instancia local de esta clase a la que solicitará servicios de resolución. Cada instancia se encargará de resolver la ubicación solicitada comunicándose con otras instancias remotas, consultando las localizaciones proporcionadas por los attachments y bindings.

En la Figura 42 se muestra el diagrama de clases correspondientes al diseño de la clase *MiddlewareSystem*. Se han omitido gran parte de los métodos privados y aquellos exclusivos de la implementación, con la finalidad de que el diagrama sea claro.

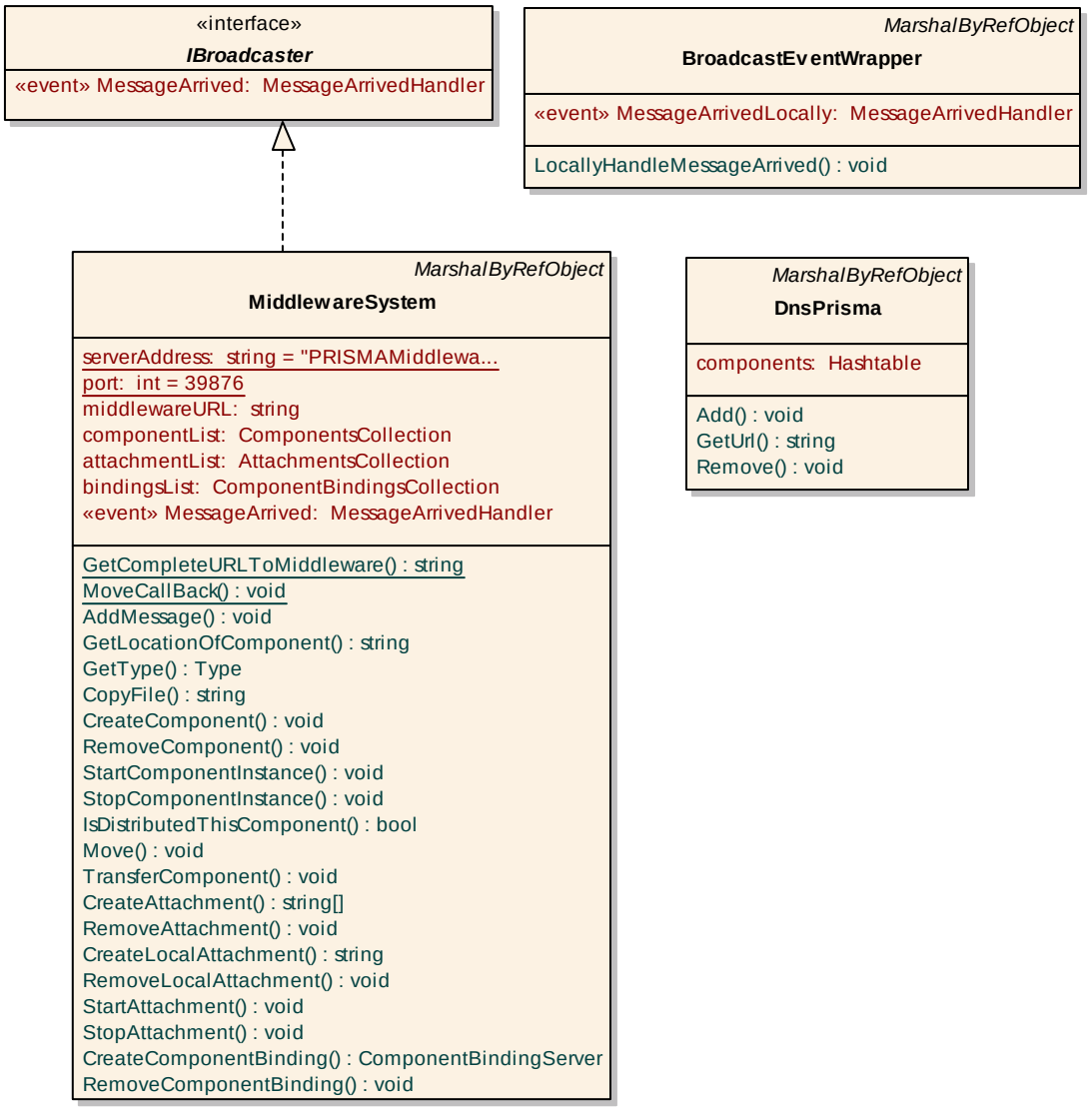

Figura 42 - Clase MiddlewareSystem

La interfaz *IBroadcaster* y la clase *BroadcastEventWrapper* permiten implementar el mecanismo de suscripción de eventos del *middleware*. Esto se ha realizado con la finalidad de que un único nodo pueda administrar la información de todos los *middleware* activos. La interfaz *IBroadcaster* define el evento que deben implementar los *middleware* para que se suscriban los posibles clientes. El *middleware* envía los mensajes de depuración a los suscriptores del evento *MesssageArrived*, y para soportar el paso de eventos en entornos distribuidos, Remoting requiere de una clase auxiliar accesible remotamente (*BroadcastEventWrapper*) que reenvíe la petición a los suscriptores. De esta forma, la aplicación de administración de PRISMA (*Prisma-Framework*) cuando es creada se registra en todos los middleware PRISMA, con lo que recibe todos los mensajes de depuración para que el arquitecto de modelos PRISMA (o el administrador) realicen las operaciones pertinentes.

La clase *MiddlewareSystem* define como atributos estáticos los parámetros para conectarse a ella. Éstos son el puerto de la máquina en la que se registrará para su acceso remoto (*port,* por defecto 39876) y el nombre del servidor (*serverAddress*) por el que será accedida, que por defecto es "PRISMAMiddleware.soap". Se ha implementado el método estático *GetCompleteURLToMiddleware* que, dada la URL del *middleware* al que se desea conectar, a partir de los parámetros anteriores construye la cadena de conexión necesaria para acceder remotamente a ella. Internamente, la clase almacena la URL donde se encuentra ubicada en el atributo *middlewareURL*.

La clase contiene tres listas: *componentList*, *attachmentList* y *bindingsList*. En la lista *componentList* se almacenan las referencias a todos los componentes, conectores y sistemas que son gestionados por este *middleware*. Del mismo modo, en *attachmentList* y *bindingsList* se almacenan las referencias correspondientes a los attachments y bindings que residen en este *middleware*. Los componentes, conectores y sistemas cuando son creados, son registrados en el DNS, para que otros *middleware* conozcan de su existencia. En cambio, los attachments y bindings no son registrados, ya que son entidades auxiliares para establecer la comunicación y únicamente deben conocerlos los *middleware* implicados. Sin embargo, a partir del nombre de un attachment o binding puede extraerse el nombre de los componentes que conecta y, como éstos si que están registrados en el DNS, puede obtenerse la localización de los attachments asociados.

A continuación se describen brevemente los servicios que proporciona el *middleware*, agrupados por la funcionalidad ofrecida. Los métodos ofrecidos por el *middleware* pueden ser invocados indistintamente desde cualquier máquina. Será tarea del *middleware* que ha recibido la petición el averiguar a qué *middleware* concreto se le debe redirigir dicha petición. Por ejemplo, puede invocarse desde un "nodo A" la creación de un attachment entre los componentes que residen en el "nodo B" y el "nodo C". El *middleware* se encargará de reenviar las peticiones adecuadas a cada nodo para realizar la tarea solicitada.

El método *AddMessage(string message)* es invocado por los tipos PRISMA en ejecución para registrar cualquier evento a considerar, como por ejemplo, la creación de componentes, la inserción de aspectos, etc. La invocación de este método propaga el mensaje a los suscriptores del evento *messageArrived*.

El método *GetLocationOfComponent(string componentName)* devuelve la ubicación del componente, conector o sistema a partir del nombre proporcionado. Para ello, propaga la petición al DNS, el cual se encarga de obtener el resultado.

Por su parte, el método *GetType(string TypeName)* se encarga de la búsqueda del tipo cuyo nombre se proporciona como parámetro. Por ejemplo, este método es invocado en la creación dinámica de puertos, attachments o bindings, ya que requieren un tipo específico a la interfaz para la que se

crean. Para ello, en primer lugar busca el tipo en los ensamblados locales, almacenados en la librería de tipos PRISMA que debe residir en una subcarpeta del directorio de trabajo del *middleware*. En segundo lugar, si no lo encuentra, va interrogando a los diferentes *middleware*, hasta encontrar uno que sí lo tenga. Es entonces cuando se invoca el método *CopyFile* para iniciar la transferencia del ensamblado que contiene el tipo hacia el *middleware* que lo ha solicitado.

La gestión de los componentes es realizada a través de varios servicios. El método *CreateComponent(Type componentType, object[] args, string URL)* es el encargado de crear un componente, conector o sistema. Para ello, se le debe proporcionar el tipo de componente a crear, los argumentos requeridos por el constructor y la URL donde se ubicará el componente. Si la localización se refiere a otra máquina distinta, la solicitud de creación es reenviada al *middleware* ubicado en dicha localización. En segundo lugar, el componente es creado a partir del tipo, registrado en el DNS y añadido a la lista interna de componentes. El método *RemoveComponent* realiza el proceso contrario. Sin embargo, un componente no inicia su funcionamiento hasta que es invocado el método *StartComponentInstance(string name)*. Este método crea un nuevo hilo para el componente y lo pone en ejecución, mediante la invocación del método *Start()*. Del mismo modo, la invocación del método *StopComponentInstance(string name)* detiene la ejecución del componente cuyo nombre se indica.

La gestión de los attachments es realizada por otro conjunto de servicios. La creación de un attachment se solicita mediante la invocación del método *CreateAttachment*, al que se le proporciona el nombre del componente y conector a conectar, así como el nombre del puerto y rol. Este método invoca al método *CreateLocalAttachment* en cada *middleware* donde residen los componentes a conectar, con la finalidad de que creen los dos attachments correspondientes. La eliminación de attachments se realiza de un modo similar mediante la invocación del método *RemoveAttachment*, que a su vez invoca los métodos *RemoveLocalAttachment* en los *middlewares* donde resida cada parte del attachment. Los attachments creados son puestos en ejecución a través de la invocación del método *StartAttachment(string name)* y son detenidos por el método *StopAttachment(string name)*.

Los bindings son creados a través de la clase *SystemBinding*, descrita en el apartado 4.2.5.3. Sin embargo, la creación remota de los *ComponentBindings* debe ser realizada por el *middleware*. Este método se limita a crear el *componentBinding* cuyos datos se proporcionan como parámetros y añadirlo a la lista interna de bindings, devolviendo un proxy hacia su parte servidora, para que pueda ser accedido remotamente. Por su parte, el método *RemoveComponentBinding* lo elimina y lo borra de la lista interna.

Los servicios de movilidad son proporcionados por los métodos *Move*, *MoveCallBack* y *TransferComponent*. El método *Move(LOC newLocation,*

string componentName) es invocado para mover un componente, conector o sistema en ejecución a otra máquina, sin perder el estado, y continuar su ejecución en la nueva ubicación. La ejecución de este método implica una serie de pasos antes de poder transferir al componente hasta su nueva ubicación, los cuales se describen aquí de forma general:

1. Se obtienen todos los attachments y bindings que están conectados a los puertos del componente a mover, para posteriormente procesarlos.
2. Se invoca el método *Stop* del componente para que realice una parada segura, lo que implica parar recursivamente también los aspectos que lo componen.
3. Una vez el componente ha sido detenido, son detenidos también los attachments y bindings a él conectados.
4. Se procesan los attachments y bindings locales con dos motivos: los que estaban publicados para su acceso remoto, son desregistrados, pues van a moverse a otra máquina. Por otra parte, si el attachment o binding que se va a mover era local (es decir, sus dos partes estaban en la misma máquina), es necesario publicar para acceso remoto la parte del attachment/binding que no se mueve, con la finalidad de que posteriormente pueda comunicarse con la parte remota (la que va a ser movida).
5. Se borra del DNS al componente que va a ser movido
6. Se transfiere al *middleware* remoto el componente a mover, junto a la lista de attachments y bindings asociados, mediante la invocación del método *TransferComponent*.
7. Se elimina el componente, los attachments y los bindings del *middleware* local que han sido transferidos a la nueva ubicación.
8. Si ocurre algun error durante el proceso, todas las acciones son deshechas.

El método *TransferComponent* también forma parte del proceso de movilidad, y realiza las operaciones necesarias en el *middleware* destino para poner en ejecución todos los elementos transferidos. En primer lugar deserializa el componente recibido y lo pone en ejecución. En segundo lugar deserializa los attachments, reestablece las comunicaciones entre los attachments recibidos y los remotos, y finalmente los pone en ejecución. Con los bindings recibidos se procede de igual manera. De esta forma, se consigue mover un componente en ejecución desde una máquina a otra y restablecer las comunicaciones, gracias a la parada segura proporcionada por los elementos encargados de las comunicaciones.

El método *MoveCallBack* se proporciona para permitir la llamada al método *Move* de forma asíncrona. Esto es necesario para aquellos casos en que un componente indique al *middleware* que desea moverse a petición propia (por ejemplo, como ocurre con el aspecto de distribución). Si la llamada no fuese asíncrona, cuando el componente invocase el servicio de movimiento, se produciría un interbloqueo: el método *Move* no puede continuar su proceso

hasta que el componente a mover sea parado, pero dicho componente no puede pararse hasta que el método que él ha invocado (*Move*) finalice su ejecución, por lo que se produce un ciclo del que no se finalizará nunca.

4.3 Resumen del Capítulo

Finalmente, a modo de resumen, se muestran las correspondencias entre los diferentes tipos PRISMA y las clases implementadas en C#:

MODELO PRISMA	IMPLEMENTACIÓN C#
Interfaz	Interfaz
Aspectos	Clase Aspecto (*AspectBase)*
Componentes	Clase Componente (*ComponentBase)*
Conectores	Clase Componente e interfaz específica (*IConnector*)
Attachments	2 instancias de la clase Attachment (*AttachmentBase)*
Binding	2 clases: una para el sistema (*SystemBinding*) y otra para el componente (*ComponentBinding*)
Sistema	Clase Sistema (*SystemBase)*

Tabla 7 - Resumen correspondencias PRISMA y la implementación C#

ASPECTO

MODELO PRISMA	IMPLEMENTACIÓN C #
Tipo	Representado a través de la herencia de una clase que representa al tipo deseado. Esta clase a su vez hereda el comportamiento básico del aspecto.
Nombre	Variable *aspectName*
Aspecto	Clase base (*AspectBase*) con lista de *played_roles* y un hilo de ejecución para procesar las peticiones de servicio. Los atributos y servicios no son contemplados, son implementados por el compilador.
Atributos	Variables
Servicios	Dos métodos, uno externo para encolar la petición de servicio y devolver el control al componente y otro interno para ejecutar el servicio solicitado.
Valuaciones	Implementación del método interno al que está asociada la valuación.
Precondiciones	Comprobadas a través de una condición dentro del cuerpo del método interno, previa a la ejecución de la valuación.
Played_Roles	Clase *SubProcess*, que contiene el nombre del played_role y el conjunto de métodos y sus propiedades.
Protocolo	El conjunto de estados se almacena en una enumeración. Es comprobado en el cuerpo del método interno antes que las precondiciones. El nuevo estado es asignado al final del método si la ejecución ha sido correcta.

Tabla 8 - Correspondencias Aspecto PRISMA y la implementación C#

COMPONENTE

MODELO PRISMA	IMPLEMENTACIÓN C #
Nombre	Variable *Name*
Interfaz	Conjunto de servicios que invoca el middleware para la gestión de componentes (*IComponent*)
Componente	Clase base con referencias a aspectos, puertos, weavings, un thread propio para la gestión de servicios, y una cola de peticiones de servicios. (*ComponentBase*)
Puertos	Colección de puertos del componente. Se divide en dos tipos: - Cliente (*OutPort*): deja las peticiones en una cola escuchada por el attachment - Servidor (*InPort*): redirecciona las peticiones a la cola del componente.
Aspectos	Lista dinámica de aspectos (*aspectList*)
Weavings	Clase *Weaving Collection* = Lista de Weavings asociada a una componente. Clase *Weaving Node* = Lista de weavings asociados al servicio que desencadena el weaving. Clase *Weaving Method* = Clase que contiene el servicio no desencadenante del weaving y las funciones de transformación necesarias para los parámetros del weaving. Clase *Weaving Type* = Tipos de weavings. Delegados hacia funciones que permiten la transformación de parámetros.

Tabla 9 - Correspondencias Componente PRISMA y la implementación C#

CONECTOR

MODELO PRISMA	IMPLEMENTACIÓN C #
Nombre	Variable *Name*
Interfaz	Conjunto de servicios que invoca el middleware para la gestión de conectores (*IConnector*, subclase de *IComponent*)
Conector	Clase *ComponentBase* que se identifica como conector por el hecho de implementar la interfaz del conector.
Roles	Colección de puertos (como el componente)
Aspectos	Colección de aspectos (como el componente)
Weavings	Colección de weavings (como el componente)

Tabla 10 - Correspondencias Conector PRISMA y la implementación C#

SISTEMA

MODELO PRISMA	IMPLEMENTACIÓN C #
Nombre	Variable *Name*
Interfaz	Conjunto de servicios que invoca el middleware para la gestión de sistemas (*ISystem*, subclase de *IComponent*)
Sistema	Clase *SystemBase*, subclase del componente, que contiene la lista de componentes, conectores y attachments que contiene, y el conjunto de bindings.

Tabla 11 - Correspondencias Sistema PRISMA y la implementación C#

ATTACHMENT

MODELO PRISMA	IMPLEMENTACIÓN C #
Nombre	Variable *attachmentName*
Attachment	Dos clases *Attachment* asociadas a cada extremo del canal de comunicación. Cada clase está formada por dos clases agregadas, una cliente y otra servidora para procesar por canales diferentes las peticiones y las recepciones.

Tabla 12 - Correspondencias Attachment PRISMA y la implementación C#

BINDING

MODELO PRISMA	IMPLEMENTACIÓN C #
Nombre	Variable *bindingName*
Binding	Dos clases específicas, una para el sistema (*SystemBinding*) y otra para el componente o conector que lo forma (*ComponentBinding*). La clase *SystemBinding* actúa como servidor remoto para aquellos componentes o conectores que están en otra máquina. Gestiona las redirecciones desde los puertos del sistema hacia los componentes, tanto en un sentido como en el otro. La clase *ComponentBinding* implementa al conector o al componente y está compuesta por dos clases agregadas, una cliente y otra servidora, para procesar por canales diferentes las peticiones y las recepciones.

Tabla 13 - Correspondencias Binding PRISMA y la implementación C#