Diseño basado en componentes: alternativas en el paso de partición

Arantza Irastorza, Arturo Jaime, Oscar Díaz

Grupo EKIN

Dpto. Lenguajes y Sistemas Informáticos
Universidad del País Vasco / Euskal Herriko Unibertsitatea
Apdo. 649 – 20.080 Donostia
{jipirgoa, jipjaela, jipdigao}@si.ehu.es

Resumen. Recientemente han surgido diferentes métodos para el diseño basado en componentes (DBC). Estos métodos conforman un conjunto de guías en el proceso de gestación e interconexión de componentes. Este trabajo examina el enfoque que se da en algunas de estas propuestas a la etapa de partición de componentes. El resultado de las mismas se muestra para un ejemplo común. La diferencia de los resultados obtenidos pone en evidencia la necesidad de unos criterios que ayuden a decidir la alternativa más razonable en cada caso.

1. Introducción

Ante la falta de un consenso claro sobre qué es esencia y qué accesorio en la noción de componente, parece necesario empezar con una definición sobre dicha noción. En la literatura se encuentran algunas definiciones más sesgadas hacia la implementación, mientras que otras ofrecen una visión más general. Dentro de las primeras estarían las que entienden que un componente es un paquete coherente de código que (a) puede ser desarrollado y distribuido independientemente, (b) tiene interfaces explícitos y bien especificados para el servicio que ofrece, (c) tiene interfaces explícitos y bien especificados para el servicio que espera de otros componentes, y (d) puede ser compuesto junto con otros componentes, quizás adaptando alguna de sus propiedades, pero sin modificar al componente propiamente dicho [4]. En [12] se matiza que no se trata de cualquier código, sino de código binario. Una definición más general nos la ofrece [9] cuando define un componente como un artefacto que ha sido desarrollado específicamente para ser reusado. En este caso un componente podría ser tanto un caso de uso como cualquier otro elemento que surja durante el desarrollo del software.

Diseño basado en componentes se refiere a la construcción de sistemas a partir de la conexión de componentes que ya existían de antes o componentes desarrolladas *ad hoc*. Recientemente han surgido diferentes métodos para el DBC. Estos métodos conforman un conjunto de guías en el proceso de gestación e interconexión de componentes. La complejidad que puede tener el sistema a desarrollar, el posible impacto en la posterior reutilización del componente y la escasa experiencia práctica en esta área, motivan el interés de estos métodos.

En la siguiente sección presentamos los pasos que se pueden distinguir en el DBC. La sección 3, introduce un dominio de trabajo que servirá de base para explorar las diferentes propuestas de partición de un sistema en componentes, dicho estudio se presenta en la sección 4. Finalmente planteamos una serie de conclusiones.

2. Diseño basado en Componentes

A pesar de tratarse de un área de estudio relativamente reciente, el DBC cuenta en la actualidad con un conjunto de propuestas diferentes. Todas ellas plantean una serie de pasos que concluyen en la obtención de un sistema construido a partir de un conjunto de componentes interconectados. Sin embargo, los objetivos, técnicas y diagramas utilizados para su consecución distan de ser uniformes. En esta sección presentamos lo que para nosotros serían los pasos básicos del DBC, describiendo para cada uno de ellos las propuestas encontradas en la literatura.

Desde nuestro punto de vista, en el proceso del DBC se pueden distinguir cinco pasos, que no tendrían por qué desarrollarse todos en secuencia:

- **Obtención de requisitos**: consiste en la extracción y análisis de las necesidades de los usuarios respecto al sistema a ser desarrollado. Encontramos tres perspectivas diferentes: *análisis vertical, análisis horizontal y análisis específico*. El *análisis vertical* se centra en un dominio o un área de negocio concreta (por ejemplo la gestión académica), con el objetivo de que los componentes resultantes puedan convertirse en estándares para cualquier aplicación desarrollada posteriormente en ese dominio [12, 3, 10, 4, 11]. Se buscan componentes fácilmente reutilizables en diferentes entornos (por ejemplo, distintas universidades). Estos componentes se limitan a la lógica general del negocio, dejando las cuestiones "de detalle" a cada aplicación. En el *análisis horizontal* se hace un análisis genérico. El objetivo es dar servicio a un rango amplio de aplicaciones posibles, sin restringirse a un dominio de negocio dado [12, 4, 3]. Este enfoque obtiene componentes que gestionan la seguridad, las transacciones, etc. Son de propósito general. Los componentes no contienen lógica específica de un dominio concreto, y ofrecen sus servicios a componentes de más alto nivel, como los obtenidos con el análisis vertical. Por último, en el *análisis específico*, se analiza una necesidad en un dominio concreto (por ejemplo, la gestión académica en la Universidad del País Vasco) [1, 2]. De este análisis, se obtienen componentes *ad hoc*. El objetivo es que el DBC facilite la evolución de la aplicación ante nuevos requisitos.
- **Partición en componentes**: como resultado el sistema se divide en un conjunto de componentes. Algunos de ellos pueden existir ya, otros pueden construirse adaptando componentes existentes o en algunos casos tal vez sea necesario construirlos completamente. En esta etapa se realiza también la identificación de servicios de estos últimos: tanto los servicios ofrecidos a otros componentes, como los necesitados para la consecución de sus objetivos. Se deja explícito cómo se va a conectar cada uno de los componentes con los demás. La partición en componentes se puede basar en *casos de uso, patrones de diseño, dominios de negocio, cambios previstos* y *componentes existentes*. También existen diferentes criterios para identificar los servicios, como por ejemplo *casos de uso* o *patrones de diseño*.
- **Diseño interno**: consiste en el diseño de cada componente teniendo en cuenta la especificación del mismo.
- **Evolución del componente**: dada la rápida evolución del mundo de los negocios, y por lo tanto de las aplicaciones informáticas que les dan soporte, el objetivo sería anticiparse a cambios futuros que pueden preverse en el momento del análisis y diseño del componente. De esta manera, por una parte, el mantenimiento posterior del componente sería mínimo o nulo, y por otra, el componente podría ser reutilizado en otras aplicaciones.
- **Interconexión de componentes**: La aplicación se concibe como un conjunto de componentes que interactúan entre sí, ofreciendo (requiriendo) los servicios a (de) otros componentes. Esta interacción es directa (interacción simple) si el servicio ofrecido se ajusta a las formas y necesidades del servicio requerido. Si las formas no son las adecuadas, es necesario realizar previamente un proceso de envoltura (*wrapper*). Esta situación se da, por ejemplo, cuando se quiere reutilizar la funcionalidad proporcionada por un sistema legado. La funcionalidad es la que queremos, pero la forma de interactuar no es la adecuada. Para ello el sistema legado "se envuelve" dentro de un software que le da apariencia de componente [1, 9]. Si el problema no está en la forma, sino en cierto desajuste entre las necesidades solicitadas y las necesidades ofertadas, se tendrá que recurrir a la adaptación del componente. El proveedor de un componente puede haber previsto posibles cambios que puede sufrir dicho componente. El cliente (o usuario del componente) fijaría en el momento del ensamblaje el cambio concreto a establecer. La técnica más conocida para lograr este objetivo es la parametrización.

3. Dominio del Caso de Estudio

Supongamos que tenemos que diseñar un sistema de información para un banco. Las entrevistas con los usuarios y el estudio del conjunto de requisitos obtenido, nos lleva a identificar, entre otras, las siguientes entidades: cliente, cuenta corriente, préstamo. Además, tendremos los siguientes casos de uso, que quedan recogidos en la figura 1:

- *Consulta saldo*: un cliente del banco, tras identificarse y proporcionar una cuenta corriente, podrá visualizar el saldo de la misma.
- *Reintegro*: un cliente del banco, tras identificarse y proporcionar una cuenta corriente, solicitará la cantidad de dinero, que desea obtener. Si la cantidad no supera la disponible en caja, se comprueba el saldo de la cuenta. Si éste es mayor que dicha cantidad, se actualizará, y se extraerá de caja la cantidad

solicitada. Si la cantidad supera el saldo en menos de un 20%, se dará comienzo a un proceso de solicitud de préstamo por descubierto. En cualquiera de los dos casos se entregará un justificante de la operación realizada. Si la cantidad sí supera el 20% del saldo se anulará la operación de reintegro, informando convenientemente al cliente.

- *Ingreso*: un cliente del banco proporciona una cuenta corriente, indica la cantidad de dinero que desea ingresar y entrega dicha cantidad. Tras comprobar que la cantidad entregada corresponde con la indicada, se actualizará el saldo de la cuenta y el dinero se introducirá en la caja de seguridad. El cliente recibirá un justificante de la operación realizada.
- *Transferencia*: un cliente del banco, tras identificarse y proporciona dos cuentas corrientes y una cantidad de dinero a transferir. Tras comprobar la existencia de las cuentas y que el saldo de la primera es mayor que la cantidad, se actualizarán los saldos de ambas (restando la cantidad a la primera y sumándosela a la segunda). Se entregará un justificante de la operación. Si la cantidad supera el saldo en menos de un 20%, se actualizarán ambas cuentas y se entregará justificante de la operación. Además, se registrará como préstamo sobre la primera cuenta, la diferencia entre lo solicitado y lo disponible en cuenta. En caso de que la cantidad a pagar supere en más de un 20% el saldo de la cuenta, se anulará la operación de transferencia, informando convenientemente al cliente.
- *Pago con tarjeta*: un cliente del banco, tras identificarse y proporcionar la clave de acceso de la tarjeta electrónica asociada a una de sus cuentas, indicará la cantidad de dinero a pagar y una cuenta corriente. Se comprueba la existencia de esta última y si el saldo de la cuenta asociada a la tarjeta es mayor que dicha cantidad. En ese caso, se actualizarán los saldos de ambas cuentas, restando la cantidad a la primera y sumándosela a la segunda, y se entregará un justificante de la operación. Si la cantidad supera el saldo en menos de un 20%, se actualizarán ambas cuentas y se entregará justificante de la operación. Además, se registrará como préstamo sobre la cuenta asociada a la tarjeta, la diferencia entre lo solicitado y lo disponible en cuenta. En caso de que la cantidad a pagar supere en más de un 20% el saldo de la cuenta, se anulará la operación de pago, informando convenientemente al cliente.
- *Pago domiciliado*: periódicamente se realizará el pago de una factura con cargo a la cuenta de un cliente. Para ello, tras proporcionar la cuenta, la cantidad endeudada y la identificación del emisor de la factura, se procederá a realizar una transferencia, por el importe indicado, de la cuenta del cliente, a la del emisor de la factura.
- *Cálculo interés*: un administrador del banco, tras identificarse, introducirá el porcentaje del interés. Seguidamente el sistema calculará el saldo medio anual de las cuentas corrientes y les aplicará dicho interés, obteniendo la cantidad con la que debe incrementar el saldo de cada una de ellas.
- *Oferta cuenta especial*: en el momento en que una cuenta corriente sobrepasa cierto umbral, se enviará al cliente información sobre otros productos bancarios (como planes de pensiones o inversiones en bolsa).
- *Petición préstamo*: las peticiones de préstamo se gestionan de manera automática. Cuando un cliente solicite un préstamo, se comprueba su identificación y el estado de su cuenta. Los préstamos pueden ser de tres tipos: 'hipotecario', 'personal' o 'por descubierto' (cuando hacemos un gasto superior al saldo de la cuenta).

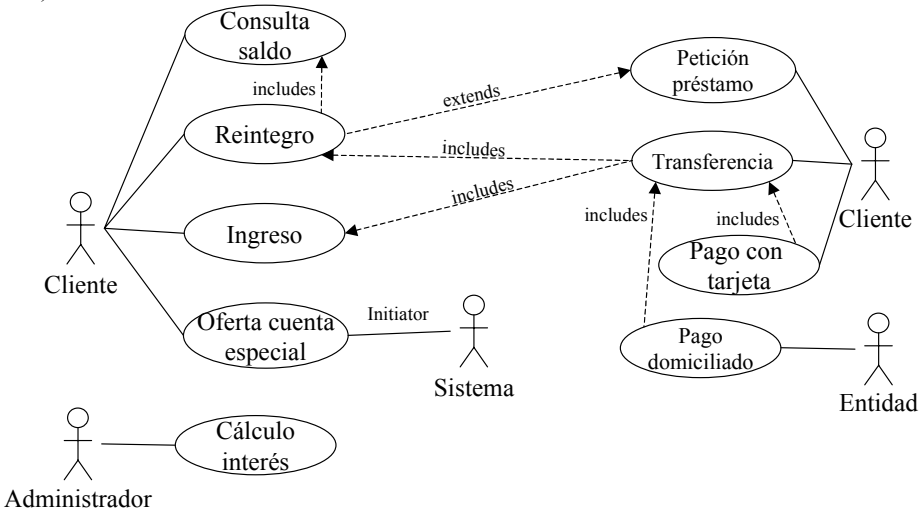

Fig. 1. Diagrama de Casos de Uso

La política de asignación de préstamos es la siguiente:

a) Cuando se solicita un préstamo, si la cuenta no tiene préstamos o si tiene uno 'por descubierto': si el préstamo pedido es 'por descubierto', se aceptará en función del saldo medio del último año; si es 'hipotecario' o 'personal', dependerá del sueldo del cliente.

b) Si la cuenta ya tiene un préstamo 'hipotecario', el nuevo préstamo sólo se aceptará si es 'por descubierto' y en función del saldo medio del último año.

c) Si la cuenta ya tiene un préstamo 'personal' no se aceptarán préstamos 'hipotecarios'. Se aceptarán préstamos 'por descubierto' en función del saldo medio del último año, y préstamos 'personales', dependiendo del sueldo del cliente.

4. Partición en componentes

Tras obtener los requisitos del sistema, la primera tarea del DBC consiste en distribuir la funcionalidad del mismo en uno o varios componentes. Alguno de dichos componentes puede existir ya o se puede conseguir adaptando un componente existente. En otro caso será necesario construir el componente desde el inicio.

Para realizar esta tarea encontramos diferentes propuestas en la literatura. El objetivo de esta sección es explicar mediante un ejemplo algunas de éstas, y comprobar que se pueden obtener particiones diferentes para el mismo problema.

En primer lugar realizaremos la partición basándonos en los casos de uso, después en los patrones de diseño, en los dominios de negocio, en los cambios previstos en el sistema y por último, teniendo en cuenta los componentes existentes.

4.1. Casos de uso

Constituyen una división del sistema basada en la funcionalidad esperada del mismo. En [9, 2, 4] se propone que la funcionalidad expresada en cada caso de uso del sistema constituya un componente separado.

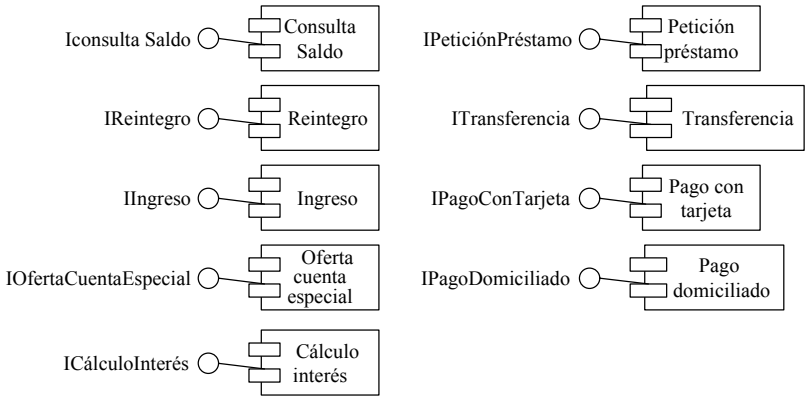

Fig. 2. Resultado de la partición con casos de uso

Un caso de uso trabajará con ciertas clases, que se identifican en el diagrama de clases. Dichas clases formarán parte del componente asociado al caso de uso. A primera vista, esta opción resulta sencilla de poner en práctica, como se muestra para nuestro ejemplo en la figura 2.

Sin embargo, debemos considerar que una misma clase puede aparecer en varios casos de uso. Por ejemplo, para los casos de uso 'Consulta saldo' y 'Reintegro' se repiten las clases *Cuenta* y *Cliente*. En 'Reintegro' e 'Ingreso', se repiten las clases *Cuenta* y *Operación*. De esta forma, y siguiendo fielmente la idea de un componente por cada caso de uso, todas las clases de un mismo caso de uso deberían ubicarse en el mismo componente. Si una clase forma parte de dos casos de uso diferentes, debería aparecer en los dos componentes correspondientes. En nuestro ejemplo tendríamos la clase *Cuenta* repetida en los componentes *Consulta saldo*, *Ingreso* y *Reintegro*.

En las propuestas estudiadas no queda claro qué conviene hacer en este caso. Si seguimos la idea de enlaces entre componentes [4] podíamos tener la clase repartida en los distintos componentes. Por ejemplo la clase *Cuenta* quedaría dividida en los componentes *Consulta saldo* y *Reintegro,* como se muestra en la figura 3. Cada componente tendría la parte de la clase *Cuenta* que necesita, aunque ello suponga repetir información de estado o funcionalidad. Además, la información de una instancia de cuenta estaría referenciada por la misma clave (el número de cuenta) en las distintas instancias de componente.

El problema radica en que no siempre es trivial repartir la funcionalidad de una clase entre varios componentes. Tomemos por ejemplo el caso de la consulta del saldo necesaria en el componente *Consulta saldo* y el decremento del saldo necesario en el componente *Reintegro.* Ambas operaciones acceden al atributo saldo, que precisaría estar replicado en ambas instancias de componente y contener el mismo valor. En el área de las bases de datos ya se ha estudiado el problema de la duplicación de información, por lo que no insistiremos aquí más sobre ello.

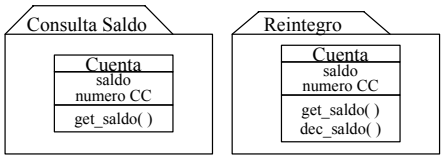

Fig. 3. Clase cuenta repartida en varias componentes

Otras posibilidades que podrían tenerse en cuenta incluyen ubicar la clase en un solo componente y que el resto de componentes la utilicen a través de los servicios proporcionados por el componente que la contiene. Así, definiríamos los componentes: *Consulta saldo*, *Ingreso* y *Reintegro* para los casos de uso del mismo nombre. El componente *Consulta saldo*, por ejemplo, contendría la clase *Cuenta* y ofrecería servicio a los otros dos (mediante las interfaces *ICuentaIngreso* e *ICuentaReintegro*). Habría una relación de dependencia de *Ingreso* y *Reintegro* hacia *Consulta saldo,* como se muestra en la figura 4 a través de las flechas en línea discontinua. Esta idea es la misma que propone [10] para partición con patrones de diseño y que explicaremos más adelante.

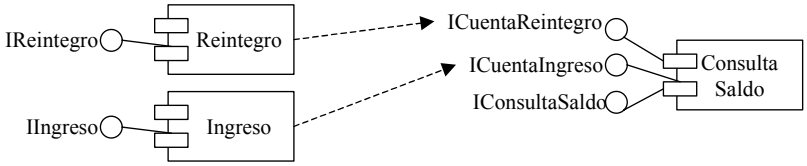

Fig. 4. Partición con casos de uso (opción 1)

Otra posibilidad es definir un componente específico que incluya la clase *Cuenta* con toda la funcionalidad necesaria en los diferentes componentes y que dé servicio a todos ellos. Esta solución se muestra en la figura 5 para el ejemplo estudiado. En este caso aparecerían componentes, como *Cuenta*, que no se corresponden directamente con un caso de uso.

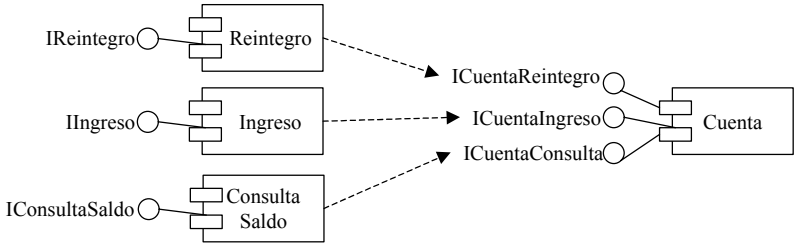

Fig. 5. Partición con casos de uso (opción 2)

4.2. Patrones de diseño

En [10] se propone que, para realizar la partición, se consideren primero los patrones que intervendrán en el diseño del sistema. La idea inicial es identificar un componente por cada patrón. Puede suceder que varios patrones incluyan clases comunes, como ocurría con los casos de uso. Siendo así, las clases de algún patrón pueden quedar distribuidas en varios componentes.

Siguiendo con nuestro ejemplo, el diseñador puede decidir utilizar los patrones estructurales *Account*, *Party*, *Contract* y los patrones de comportamiento *Observer* y *State*.

El patrón *Account* [5] de la figura 6 incluye las características de una cuenta, y los casos de uso 'Ingreso', 'Reintegro', 'Transferencia', 'Pago con Tarjeta', 'Pago domiciliado', y 'Cálculo interés'.

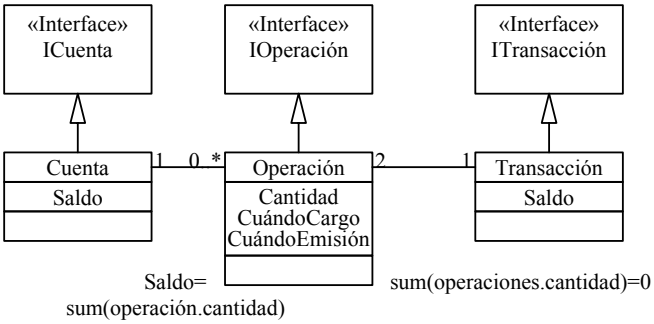

Fig. 6. Patrón de Diseño *Account*

El patrón *Party* [7] de la figura 7 incluye lo referente a clientes de las cuentas corrientes, tanto personas individuales como empresas. Utilizando el patrón *Contract* [7] de la figura 7, se diseñarían los préstamos, entendiendo que un préstamo es un contrato que firma el titular de una cuenta corriente y donde la cuenta corriente queda asociada a dicho préstamo.

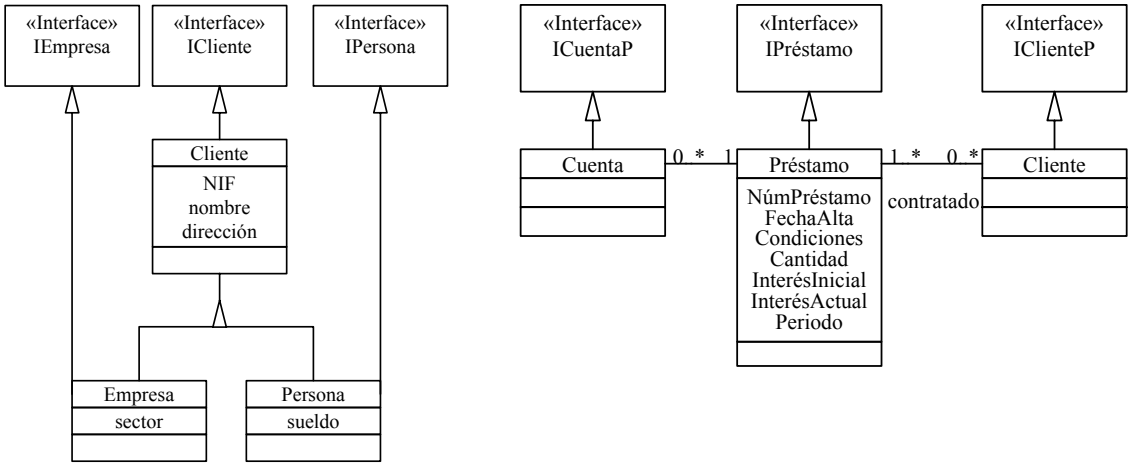

Fig. 7. Patrones de Diseño *Party* (izquierda) y *Contract* (derecha)

El caso de uso 'Oferta cuenta especial' se encarga de hacer ofertas especiales a cuentas que cumplan ciertas condiciones. En su diseño se puede utilizar el patrón *Observer* [6] de la figura 8, de forma que una cuenta corriente será observada por otra clase, que llamaremos 'Monitor'. Esta clase será la encargada de hacer el control del saldo de las cuentas cada vez que se haga un ingreso o una transferencia sobre las mismas, y de tomar la decisión correspondiente cuando se cumplan las condiciones predefinidas.

El patrón *State* [6] de la figura 8, es utilizado para el diseño del caso de uso 'Petición Préstamo'. Como se ha comentado anteriormente, este caso de uso aceptará la petición de un préstamo dependiendo de la cuenta corriente asociada al mismo. Es decir, el comportamiento del caso de uso 'Petición Préstamo' variará dependiendo del estado de la cuenta. Por ello, este patrón propone definir la clase 'EstadoCuenta', con una relación 1:1 con 'Cuenta' y varias subclases, tantas como estados pueda tener una cuenta corriente.

Siguiendo el criterio de partición basado en patrones de diseño comentado anteriormente, los patrones *Account* y *Party*, con todas sus clases darán lugar a los componentes *Cuenta* y *Cliente*, respectivamente. Además, las interfaces del primer componente serán *Icuenta*, *Ioperación* e *Itransacción*. Las del segundo serán *Icliente*, *Ipersona* e *Iorganización*. Las interfaces correspondientes a las clases incluidas en un componente configuran las interfaces del mismo.

El patrón *Contract* de la figura 7 da lugar al componente *Préstamo*. *Contract* tiene tres clases: *Cliente*, *Cuenta* y *Préstamo*. Esta última clase formaría parte del componente *Préstamo* y su interfaz *Ipréstamo* sería una interfaz del mismo. La clase *Cuenta* ya aparece en el componente *Cuenta*. En lugar de duplicar

dicha clase en ambos componentes, en el componente *Cuenta* añadiremos la interfaz *IcuentaP*. Así, el componente *Cuenta* ofrece los servicios de la clase del mismo nombre, precisados por el componente *Préstamo*, a través de dicha interfaz *IcuentaP* (se muestra en la figura 9 mediante las flechas con líneas discontinuas). La clase *Cliente* estaría en el mismo caso, ya aparece en el componente *Cliente*, y por ello únicamente añadimos a éste la interfaz *IclienteP*, que correspondía a la clase *Cliente* en el patrón *Party*.

En los patrones *Observer* y *State* de la figura 8 se produce la misma circunstancia para la clase *Cuenta*. Del mismo modo que antes, se introducen las interfaces *Imonitorizado* e *IcuentaEst* en el componente *Cuenta* para dar servicio a los componentes *Monitor* y *EstadoCuenta* respectivamente. El resto de las clases e interfaces formarán los componentes *Monitor* y *EstadoCuenta* como se muestra en la figura 9.

La figura 9 recoge el diagrama de componentes resultante de este proceso de partición.

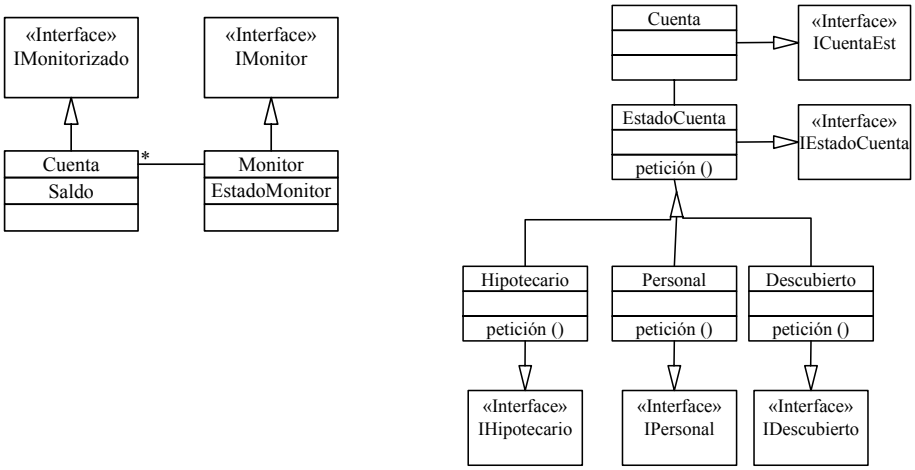

Fig. 8. Patrones de Diseño *Observer* (izquierda) y *State* (derecha).

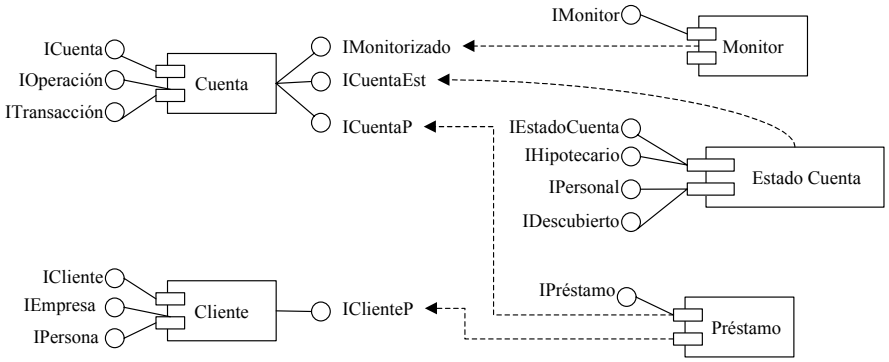

Fig. 9. Resultado de la partición con patrones de diseño

4.3. Dominios de negocio

En [1] se define dominio de negocio como el proveedor de servicios clave para los procesos de negocio. Un dominio de negocio debe representar una parte de la infraestructura de la organización, contener recursos clave, ofrecer servicios de negocio y encapsular políticas de negocio genéricas. Cada dominio de negocio da lugar a un componente. Otros autores proponen hacer la división basándose en el dominio, desde una perspectiva más cercana a la identificación de clases o tipos abstractos de datos [9].

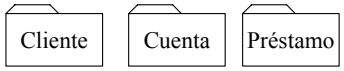

Fig. 10. Componentes relacionados con los dominios de negocio

Considerando las características anteriores, en nuestro ejemplo distinguiríamos los dominios de negocio *Cuenta*, *Cliente* y *Préstamo* (figura 10).

Los servicios ofrecidos por cada dominio se identifican a partir de los casos de uso. En primera instancia se hace un esbozo de los diagramas de clases de cada dominio. En nuestro ejemplo los diagramas de clases para los dominios detectados se muestran en la figura 11.

Seguidamente se definen los diagramas de estado para las clases y se estudian los casos de uso mediante diagramas de secuencia y diagramas de colaboración. En la figura 12, mostramos el diagrama de colaboración del caso de uso 'Petición Préstamo'. Las cajas con línea doble simbolizan dominios externos. La notación :A::B indica la clase B del componente A.

Analizando el diagrama de colaboración detectamos que el caso de uso 'Petición Préstamo' utiliza servicios de los componentes *Cuenta* y *Cliente*. Así se han identificado los servicios *identificar* y *getSueldo* del componente *Cliente* y los servicios *getEstado* y *getSaldoMedio* del componente *Cuenta*.

Los servicios que se van detectando en este proceso iterativo se añaden a las clases. Así a partir de los diagramas de clases de los componentes de la figura 11 se obtienen los nuevos diagramas de la figura 13. El diagrama de componentes resultante, considerando únicamente este caso de uso, se muestra en la figura 14. El proceso seguiría con el resto de casos de uso.

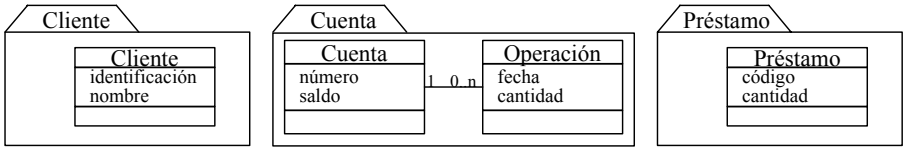

Fig. 11. Diagramas de clases de los componentes

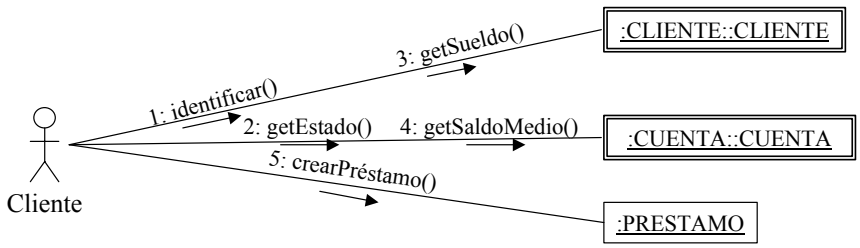

Fig. 12. Diagrama de colaboración del caso de uso 'Petición Préstamo'

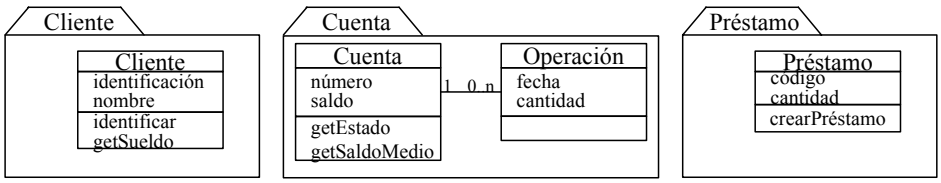

Fig. 13. Diagramas de clases modificados de los componentes

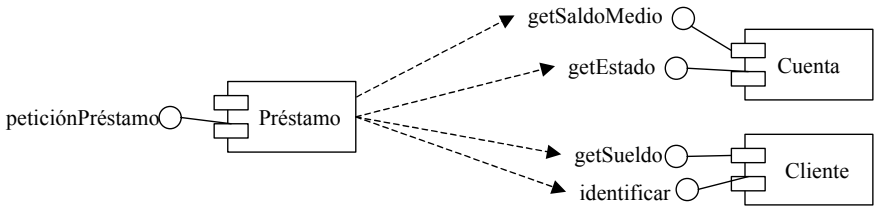

Fig. 14. Resultado (parcial) de la partición con dominios de negocio

4.4. La evolución prevista del sistema

Con esta propuesta, durante el análisis se contemplan no sólo los requisitos actuales sino también las posibles necesidades futuras. El objetivo es facilitar la evolución del sistema ante los cambios previstos. Éste es el criterio seguido en [8] con el objetivo de que un cambio afecte a un único componente.

En el ejemplo anterior, se prevén los siguientes cambios:

- Moneda de peseta a euro.
- Gestión en la comisión para cuentas cuyo saldo sea inferior a una cierta cantidad.
- Introducción de tarjetas monedero.
- Gestión del préstamo por descubierto.
- Banca electrónica.

Antes de realizar la partición en componentes, es preciso identificar las clases que intervienen en el sistema. El método estudia las operaciones y datos definidos sobre dichas clases y qué cambios de los anteriores afectan a cada uno. Todas las operaciones y datos afectados por un mismo cambio quedan agrupados en un mismo componente.

De acuerdo con esto, los cinco cambios anteriores darían lugar en principio a cinco componentes. Sin embargo, puede suceder que una misma operación, como *identificación de cliente,* se vea afectada por varios cambios, como el de la introducción de tarjeta monedero y el de la banca electrónica. Por lo tanto todas las operaciones y datos afectados por ambos cambios deberán incluirse en un solo componente.

Haciendo un estudio similar para el resto de los cambios, llegamos a agrupar las operaciones y datos involucrados, en los componentes *Cuenta* y *Transferir dinero* como se muestra en la figura 15. El componente *Cuenta* incluye las operaciones y datos afectados por los dos primeros cambios previstos y *Transferir dinero* los de los otros tres.

Siguiendo esta orientación no queda muy claro cómo distribuir en componentes todas aquellas operaciones y datos que no se vean afectadas por ninguno de los cambios previstos.

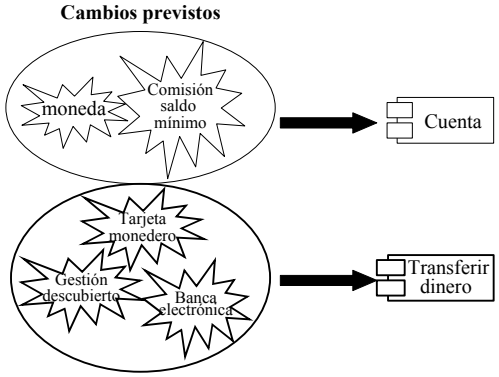

Fig. 15. Resultado (parcial) de la partición con cambios previstos

4.5. Componentes ya existentes

A diferencia de los enfoques anteriores, no se parte de cero, sino que puede haber partes del sistema que estén ya implementadas. El diseñador tiene que integrar dicho código dentro de su diseño. En ocasiones este código incluye un conjunto de componentes. Éstos pueden haberse desarrollado en la propia organización o haberse comprado a una empresa externa. En cualquier caso, los componentes pueden requerir ser adaptados. Por ello, en el segundo caso habrá que tener en cuenta que existe el riesgo de que la empresa proveedora no facilite ese tipo de servicio o que incluso haya desaparecido [13].

En nuestro ejemplo, tendremos que buscar componentes relacionados con aplicaciones bancarias que nos puedan ser de utilidad.

5. Conclusiones

Tras identificar los pasos del DBC, el objetivo de este trabajo ha sido explicar las diferentes propuestas de partición de un sistema en componentes. Hemos utilizado un ejemplo común para mostrar dichas propuestas y sus resultados finales. Es evidente que realizando la partición basándonos en la evolución prevista del sistema mejoraremos el mantenimiento del sistema resultante. Si podemos contar con componentes ya existentes, estaremos desarrollando el sistema reutilizando partes ya desarrolladas, mejorando a priori la productividad. Sin embargo, el resto de las propuestas no dejan claro qué ventajas aportan. Pensamos que sería conveniente disponer de un conjunto de criterios que permita seleccionar la propuesta más adecuada a cada caso.

Algunas propuestas de partición no dejan claros todos los casos que nos podemos encontrar. Por ejemplo, en la partición por casos de uso, no se determina cómo repartir las clases en los componentes obtenidos, cuando dichas clases participan en más de un caso de uso.

En principio, de las propuestas no se deduce que se puedan identificar los componentes a partir de criterios de partición diferentes, por ejemplo unos componentes por casos de uso y otros por dominios de negocio. En el caso de tomar la decisión de usar componentes existentes, es evidente que tendremos que usar otras propuestas para desarrollar el resto de componentes, lo que nos lleva a mezclar propuestas de partición.

Respecto a la identificación de servicios de cada componente, observamos cómo cada propuesta de partición presenta su propia técnica. Pensamos que alguna de estas técnicas puede usarse en más de una propuesta de partición. Por ejemplo, en la alternativa basada en dominios de negocio, la identificación de servicios podría basarse en la misma técnica que la propuesta para patrones de diseño.

Aunque este trabajo se ha centrado en el paso de partición en componentes, se puede realizar un estudio similar para otros pasos del DBC, como son la evolución e interconexión de componentes.

Bibliografía

[1] P.Allen y S.Frost. Component-Based Development for Enterprise Systems. Applying the SELECT Perspective™.Cambridge University Press, 1998.

[2] B.Berkem. Traceability management from business processes to use cases with UML. A proposal for extensions to the UML's activity diagram through goal-oriented objects. *Journal of Object-Oriented Programming,* páginas 29-34, 1.999.

[3] K.Bohrer, V. Johnson, A. Nilsson, B.Rubin. Business process components for distributed object applications. *Communications of the ACM*, 41(6):43-48, 1.998.

[4] D.D'Souza, A.Wills. Objects, Components and Frameworks with UML. The Catalysis approach. Addison-Wesley 1.999.

[5] M. Fowler. Analysis Patterns. Reusable Object Models. Addison-Wesley 1.997.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[7] D.C. Hay. Data Model Patterns. Conventions of Thought. Dorsert House, 1996

[8] C.L.Hoover, P.K.Khosla. Analytical partition of software components for evolvable and reliable mems design tools. *Int. Journal of software Engineering and Knowledge Engineering,* 9(2):153-171, 1.999.

[9] I. Jacobson, M Griss, P. Jonsson. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley 1.997.

[10] G.Larsen. Designing component-based frameworks using patterns in the UML. *Communications of the ACM*, 42(10):38-45, 1.999.

[11] R. van Ommering, F. van der Linden, J.Kramer, J.Magee. The Koala component model for consumer electronics software. *Computer*, páginas 78-85, 2000.

[12] C.Szyperski. Component Software. Beyond Object-Oriented Software. Addison-Wesley, 1998.

[13] J. Voas. Maintaining Component-Based Systems. *IEEE Software*, July/August, páginas 22-27, 1.998.