Introducción

Un sistema orientado a objetos se compone de objetos que envían mensajes a otros objetos para que lleven a cabo operaciones. En los contratos se incluye una conjetura inicial sobre las responsabilidades y las poscondiciones de las operaciones. Los diagramas de interacción describen, de manera gráfica, la solución a partir de los objetos en interacción que estas responsabilidades y poscondiciones satisfacen.

Las decisiones poco acertadas respecto a la asignación de responsabilidades, dan origen a sistemas y componentes frágiles y difíciles de mantener, entender, reutilizar o extender. Una buena implementación, se basa en los principios que rigen un buen diseño orientado a objetos. En los patrones, que se aplican al preparar los diagramas de interacción, se codifican algunos de estos principios cuando se asignan las responsabilidades o durante ambas actividades.

Patrones de diseño para asignar responsabilidades

Los principios del diseño que se requieren para construir buenos diagramas de interacción se pueden codificar, explicar y utilizar de forma metódica. Esta manera de entender y usar los principios de diseño, se basa en los *patrones con que se asignan las responsabilidades*.

Responsabilidades y métodos

Booch y Rumbaugh definen la responsabilidad como “un contrato u obligación de un tipo o clase”. Las responsabilidades se relacionan con las obligaciones de un objeto respecto a su comportamiento. Esas responsabilidades pertenecen, esencialmente a dos categorías: Hacer y conocer

Entre las responsabilidades de un objeto relacionadas con *hacer* se encuentran:

- Hacer algo en uno mismo.
- Iniciar una acción en otros objetos.
- Controlar y coordinar actividades en otros objetos.

Entre las responsabilidades de un objeto relacionadas con *conocer* se encuentran:

- Estar enterado de los datos privados encapsulados.
- Estar enterado de la existencia de objetos conexos.
- Estar enterado de cosas que se pueden derivar o calcular.

Las responsabilidades se asignan a los objetos durante el diseño orientado a objetos. Por ejemplo, puede declararse que “una *Venta* es responsable de imprimirse ella misma” (un hacer) o que “una *Venta* tiene la obligación de conocer su fecha” (un conocer). Las responsabilidades relacionadas con “conocer” a menudo pueden inferirse del modelo conceptual por los atributos y asociaciones explicadas en él.

Responsabilidad no es lo mismo que método: los métodos se ponen en práctica para cumplir con las responsabilidades. Éstas se implementan usando métodos que operan solos o en colaboración con otros métodos y objetos.

En UML, las responsabilidades (implementadas como métodos) se suelen tener en cuenta en el momento de elaborar los diagramas de interacción. Estos diagramas muestran las decisiones sobre la asignación de responsabilidades entre los objetos, que se reflejan en los mensajes que se envían a las clases.

Patrones de diseño

El patrón es una descripción de un problema y la esencia de su solución, de manera que esa solución se puede reutilizar en diferentes configuraciones. El patrón no es una especificación detallada. Más bien, se puede considerar como una descripción de experiencia acumulada, una solución bien probada a un problema común.

Los patrones de diseño se asocian usualmente con el diseño orientado a objetos. Los patrones publicados se suelen apoyar en características de objetos como herencia y polimorfismo para dar generalidad. Sin embargo, el principio universal de encapsular la experiencia en un patrón es igualmente aplicable a cualquier tipo de diseño de software. Los patrones son una forma de reutilizar el conocimiento y la experiencia de otros diseñadores.

Los elementos esenciales de los patrones de diseño (Gamma y sus coautores), son:

1. Un nombre que sea una referencia significativa al patrón.
2. Una descripción del problema que enuncie cuándo se puede aplicar el patrón.
3. Una descripción de la solución de diseño, sus relaciones y responsabilidades. No es una descripción de diseño concreta, sino una plantilla para que una solución se instale en diferentes formas. Esto se puede expresar gráficamente para mostrar las relaciones entre los objetos y las clases de objetos en la solución.
4. Las consecuencias, los resultados y las negociaciones, al aplicar el patrón, lo cual ayuda al diseñador a entender si es factible usar o no el patrón en una situación particular.

Los patrones GRASP

Los patrones intentan codificar conocimiento, expresiones y principios ya *existentes*. En consecuencia, los patrones GRASP no introducen ideas novedosas; son una codificación de los principios básicos más usados.

En teoría, todos los patrones poseen nombres muy sugestivos. El asignar nombre a un patrón, a un método o a un principio ofrece las siguientes ventajas:

- Apoya el agrupamiento y la incorporación del concepto a nuestro sistema cognitivo y a la memoria.
- Facilita la comunicación.

Darle nombre a una idea compleja es un ejemplo de la fuerza de la abstracción: convierte una forma complicada en una forma simple con sólo eliminar los detalles. Por lo tanto, los patrones GRASP poseen nombres concisos como *Experto*, *Creador*, *Controlador*.

Los patrones GRASP describen los principios fundamentales del diseño de objetos y la asignación de responsabilidades Es importante entender y ser capaces de aplicar estos principios durante la creación de los diagramas de interacción porque un desarrollador de software con poca experiencia en la tecnología de objetos, necesita dominar estos principios tan rápido como sea posible, puesto que constituyen la base de cómo se diseñará el sistema.

A continuación, se presentarán los siguientes patrones GRASP: *Experto en Información*, *Creador*, *Bajo Acoplamiento*, *Alta Cohesión* y *Controlador*.

Experto en información (o experto)

Problema

¿Cuál es un principio general para asignar responsabilidades a los objetos?

Un modelo de diseño podría definir cientos o miles de clases, y una aplicación podría requerir que se realicen cientos o miles de responsabilidades. Durante el diseño orientado a objetos, cuando se definen las interacciones entre los objetos, se toman decisiones sobre la asignación de responsabilidades a las clases. Si se hace bien, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y existen más oportunidades para reutilizar componentes en futuras aplicaciones.

Solución

Asignar responsabilidades al experto en información, es decir, a la clase que tiene la información necesaria para realizar la responsabilidad.

Ejemplo (ver transparencia 7)

En este ejemplo, algunas clases necesitan conocer el total de una venta.

Se comienza asignando responsabilidades con una definición clara de ellas.

Siguiendo este consejo, la pregunta es:

¿Quién debería ser el responsable de conocer el total de una venta?

Desde el punto de vista del patrón *Experto en Información*, se debería buscar la clase de objetos que contiene la información necesaria para determinar el total. Examinemos con detenimiento el modelo conceptual de la transparencia 7

¿Qué información se necesita para determinar el total? Es necesario conocer todas las instancias de *LíneaDeVenta* de una venta y la suma de sus subtotales. Una instancia de *Venta* las contiene, por lo tanto, la clase *Venta* es adecuada para esta responsabilidad; es un *experto en información.*

Es en el contexto de la creación de los diagramas de interacción (en este caso, diagrama de comunicación) cuando surgen estas cuestiones de responsabilidad.

¿Qué información se necesita para determinar el subtotal de una línea de producto? Se necesitan *LíneaDeVenta.cantidad* y *Producto.precio*, puesto que *LíneaDeVenta* conoce el *Producto* asociado y su cantidad; en consecuencia, la *LíneaDeVenta* debería calcular el subtotal; es el experto en información.

En los diagramas de interacción, esto significa que la *Venta* necesita enviar el mensaje *getSubtotal* a cada una de las *LíneaDeVenta* y sumar el resultado.

Para realizar la responsabilidad de conocer y determinar el subtotal, una *LíneaDeVenta* necesita conocer el precio del producto.

El *Producto* es un experto en información que proporciona su precio; por tanto, se le debe enviar un mensaje solicitándolo.

Discusión

El Experto en Información se utiliza con frecuencia en la asignación de responsabilidades; es un principio de guía básico que se usa continuamente en el diseño orientado a objetos. El Experto expresa la “intuición” común de que los objetos hacen las cosas relacionadas con la información que tienen.

El cumplimiento de la responsabilidad, a menudo, requiere información que se encuentra dispersa en diferentes clases. Esto implica que hay muchos expertos en información “parcial” que colaborarán en la tarea. Por ejemplo, el problema del total de una venta al final requiere la colaboración de tres clases. Cada vez que la información se encuentre esparcida por objetos diferentes, necesitarán interactuar mediante el paso de mensajes para compartir trabajo.

Contraindicaciones (consecuencias malas)

En algunas ocasiones, la solución que sugiere el Experto no es deseable, debido normalmente a problemas de acoplamiento y cohesión.

Por ejemplo, ¿quién debería ser el responsable de almacenar una *Venta* en una base de datos? Ciertamente, mucha de la información que se tiene que almacenar se encuentra en el objeto Venta y, por tanto, siguiendo el patrón Experto se podría argumentar la

inclusión de la responsabilidad en la clase *Venta*. La extensión lógica de esta decisión es que cada clase contiene sus propios servicios para almacenarse ella misma en una base de datos. Pero esto nos lleva a problemas de cohesión, acoplamiento y duplicación. Por ejemplo, la clase *Venta* debe ahora contener la lógica relacionada con la gestión de la base de datos. La clase ya no está centrada únicamente en la lógica de la aplicación de "ser una venta" simplemente; ahora tiene otro tipo de responsabilidades, lo cual disminuye su cohesión. La clase debe acoplarse a los servicios técnicos de otro subsistema, en lugar de acoplarse únicamente con los objetos en la capa del dominio de los objetos, lo que eleva su acoplamiento. Y es probable que se dupliquen lógicas de bases de datos similares en muchas clases persistentes.

Beneficios (consecuencias buenas)

- Se mantiene el encapsulamiento de la información, puesto que los objetos utilizan su propia información para llevar a cabo las tareas. Normalmente, esto conlleva un bajo acoplamiento, lo que da lugar a sistemas más robustos y más fáciles de mantener.
- Se distribuye el comportamiento entre las clases que contienen la información requerida, por tanto, se estimula las definiciones de clases más cohesivas y "ligeras" que son más fáciles de entender y mantener. Normalmente, se soporta una alta cohesión.

Creador

Problema

¿Quién debería ser el responsable de la creación de una nueva instancia de alguna clase?

La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. En consecuencia, es útil contar con un principio general para la asignación de las responsabilidades de creación. Si se asignan bien, el diseño puede soportar un bajo acoplamiento, mayor claridad, encapsulación y reutilización.

Solución

Asignar a la clase B la responsabilidad de crear una instancia de la clase A si se cumple uno o más de los casos siguientes:

- B agrega objetos de A.
- B contiene objetos de A.
- B registra instancias de objetos de A.
- B utiliza más estrechamente objetos de A.
- B tiene los datos de inicialización que se pasarán a un objeto de A cuando sea creado (por tanto, B es un Experto con respecto a la creación de A).

B es un creador de los objetos de A.

Si se puede aplicar más de una opción, inclinarse por una clase B que agregue o contenga la clase A.

Ejemplo (ver transparencia 9)

En el ejemplo, ¿quién debería ser el responsable de la creación de una instancia de *LíneaDeVenta*? Según el patrón Creador, se deberían buscar las clases que agreguen, contengan, etc. instancias de *LíneaDeVenta*.

Puesto que una *Venta* contiene muchos objetos *LíneaDeVenta*, el patrón creador sugiere que *Venta* es un buen candidato para tener la responsabilidad de crear las instancias de *LíneaDeVenta*.

Esto lleva al diseño de las interacciones (representadas en este caso por un diagrama de secuencia) entre objetos que se muestran en la transparencia 9.

Discusión

La intención básica de este patrón es encontrar un creador que necesite conectarse al objeto creado en alguna situación. Eligiéndolo como el creador se favorece el bajo acoplamiento.

A veces se encuentra un creador buscando las clases que tienen los datos de inicialización que se pasarán durante la creación. Se trata, en realidad, de un ejemplo del patrón Experto. Los datos de inicialización se pasan durante la creación por medio de algún tipo de método de inicialización. Por ejemplo, supongamos que una instancia de *Pago* necesita inicializarse cuando se crea, con el total de *Venta*. Puesto que la *Venta* conoce el total, la *Venta* es un creador candidato del *Pago*.

Contraindicaciones (consecuencias malas)

A menudo, la creación requiere una complejidad significativa, como utilizar instancias recicladas por motivos de rendimiento, crear condicionalmente una instancia a partir de una familia de clases similares basándose en el valor de alguna propiedad externa, etc.

Beneficios (consecuencias buenas)

Se soporta bajo acoplamiento, lo que implica menos mantenimiento y mayores oportunidades para reutilizar. Probablemente, no se incrementa el acoplamiento porque la clase *creada* es presumible que ya sea visible a la clase *creadora*, debido a las asociaciones existentes que motivaron su elección como creador.

Bajo acoplamiento

Problema

¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?

El *acoplamiento* es una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos. Un elemento con bajo (o débil) acoplamiento no depende de demasiados otros elementos; “demasiados” depende del contexto. Estos elementos pueden ser clases, subsistemas, sistemas, etc.

Una clase con alto (o fuerte) acoplamiento confía en muchas otras clases, que podrían ocasionar algunos de los siguientes problemas:

- Los cambios en las clases relacionadas fuerzan cambios locales.
- Son difíciles de entender de manera aislada.
- Son difíciles de reutilizar puesto que su uso requiere la presencia adicional de las clases de las que depende.

Solución

Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.

Ejemplo (ver transparencia 11)

Consideremos el diagrama de clases parcial de la transparencia 11.

Supongamos que se tiene la necesidad de crear una instancia de *Pago* y asociarla con la *Venta*. ¿Qué clase debería ser la responsable de esto? Puesto que un *Registro* “registra” un *Pago* en el mundo real, el patrón Creador sugiere el *Registro* como candidata para la creación del *Pago*. La instancia de *Registro* podría enviar entonces el mensaje *añadirPago* a la *Venta*, pasando el nuevo *Pago* como parámetro. En la transparencia 11 se muestra un posible diagrama de interacción (diagrama de comunicación) que refleja esto. También se muestra una solución alternativa a crear *Pago* y asociarlo con la *Venta*.

¿Qué diseño, basado en la asignación de responsabilidades, soporta Bajo Acoplamiento? En ambos casos se asume que la *Venta* debe finalmente acoplarse con el conocimiento del *Pago*. El Diseño 1, en el que el *Registro* crea el *Pago*, añade acoplamiento entre el *Registro* y el *Pago*, mientras que el Diseño 2, en el que la *Venta* lleva a cabo la creación del *Pago*, no incrementa el acoplamiento. Desde el punto de vista del acoplamiento, es

preferible el Diseño 2, porque mantiene el acoplamiento global más bajo. Este es un ejemplo en el que dos patrones (Bajo Acoplamiento y Creador) podrían sugerir soluciones diferentes.

En la práctica, el nivel de acoplamiento no se puede considerar de manera aislada a otros principios como el Experto o Alta Cohesión. Sin embargo, es un factor a tener en cuenta para mejorar un diseño.

Discusión

El patrón de Bajo Acoplamiento es un principio a tener en cuenta en todas las decisiones de diseño; es un objetivo subyacente a tener en cuenta continuamente. Es un *principio evaluativo* que aplica un diseñador mientras evalúa todas las decisiones de diseño.

La transparencia 12 muestra algunas de las formas comunes de acoplamiento en los lenguajes orientados a objetos tales como C++, Java y C#,

El patrón Bajo Acoplamiento no se puede considerar de manera aislada a otros patrones, sino que necesita incluirse como uno de los principios de diseño que influyen en una elección al asignar una responsabilidad.

No existe una medida absoluta de cuándo el acoplamiento es demasiado alto. Lo que es importante es que un desarrollador pueda medir el grado de acoplamiento actual y evaluar si aumentarlo causará problemas. En general, las clases que son muy genéricas por naturaleza, y con una probabilidad de reutilización alta, deberían tener un acoplamiento especialmente bajo.

El caso extremo de Bajo Acoplamiento es cuando no existe acoplamiento entre clases. Esto no es deseable porque en la tecnología de objetos, los objetos están conectados y se comunican mediante el paso de mensajes. Es normal y necesario un grado moderado de acoplamiento entre clases para crear un sistema orientado a objetos en el que las tareas se realizan mediante la colaboración de los objetos conectados.

Contraindicaciones (consecuencias malas)

El alto acoplamiento en sí no es un problema; es el alto acoplamiento entre elementos que no son estables en alguna dimensión, como su interfaz, implementación o su mera presencia.

Este es un punto importante: se puede añadir flexibilidad, encapsular implementaciones y detalles, y, en general, diseñar para disminuir el acoplamiento en muchas áreas del sistema. Pero, si nos esforzamos en “futuras necesidades” o en disminuir el acoplamiento en algunos puntos en los que no hay motivos realistas, el tiempo no se está empleando de manera adecuada.

Los diseñadores tienen que escoger sus batallas para disminuir el acoplamiento y encapsular información. Centrándose en los puntos en los que sea realista pensar en una inestabilidad o evolución alta.

Beneficios

- No afectan los cambios en otros componentes.
- Fácil de entender de manera aislada.
- Conveniente para reutilizar.

Alta cohesión

Problema

¿Cómo mantener la complejidad manejable?

En cuanto al diseño de objetos, la *cohesión* (o de manera más específica, la cohesión funcional) es una medida de la fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión. Estos elementos pueden ser clases, subsistemas, etc.

Una clase con baja cohesión hace o muchas tareas no relacionadas, o demasiado trabajo. Tales clases no son convenientes, porque son: difíciles de entender, difíciles de reutilizar, difíciles de mantener y constantemente afectadas por los cambios

Solución

Asignar una responsabilidad de manera que la cohesión permanezca alta.

Ejemplo (ver transparencia 14)

Se puede analizar el mismo ejemplo que se utilizó para el patrón Bajo Acoplamiento.

Supongamos que se necesita crear una instancia de *Pago* y asociarla con la *Venta*. ¿Qué clase debería ser responsable de esto? Puesto que el *Registro* registra un *Pago* en el mundo real, el patrón *Creador* sugiere el *Registro* como candidato para la creación de *Pago*. La instancia de *Registro* podría entonces enviar un mensaje *añadirPago* a la *Venta*, pasando como parámetro el nuevo *Pago*. Esta asignación de responsabilidades sitúa la responsabilidad de realizar un pago en el registro. El *Registro* toma parte en la responsabilidad de llevar a cabo la operación del sistema *realizarPago*.

En este ejemplo aislado esto es aceptable; pero si continuamos haciendo responsable a la clase *Registro* de realizar parte o la mayoría del trabajo relacionado con más y más operaciones del sistema, se irá sobrecargando incrementalmente con tareas y llegará a perder cohesión.

Como se muestra en transparencia 14, el segundo diseño delega la responsabilidad de la creación del pago en *Venta*, lo cual favorece una cohesión más alta en el *Registro*.

Es deseable el segundo diseño puesto que soporta alta cohesión y bajo acoplamiento.

En la práctica, el nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otros principios como los patrones Experto y Bajo Acoplamiento.

Discusión

Grady Booch establece que existe alta cohesión funcional cuando los elementos de un componente (una clase) “trabajan todos juntos para proporcionar algún comportamiento bien delimitado”.

En la transparencia 15 se muestran los distintos grados de cohesión funcional.

Como regla empírica, una clase con alta cohesión tiene un número relativamente pequeño de métodos, con funcionalidad altamente relacionada, y no realiza mucho trabajo. Colabora con otros objetos para compartir el esfuerzo si la tarea es extensa.

El acoplamiento y la cohesión son viejos principios del diseño de software; diseñar con objetos no implica que se ignoren los fundamentos bien establecidos. Otro de estos principios, fuertemente relacionado con el acoplamiento y la cohesión, es promover el *diseño modular.*

Hay que fomentar el diseño modular creando métodos y clases con alta cohesión. Al nivel básico de objetos, la modularidad se alcanza diseñando cada método con un único y claro objetivo, y agrupando un conjunto de aspectos relacionados en una clase.

Contraindicaciones (Consecuencias malas)

Existen pocos casos en los que esté justificada la aceptación de baja cohesión.

Beneficios (consecuencias buenas)

- Se incrementa la claridad y facilita la comprensión del diseño.
- Se simplifican el mantenimiento y las mejoras.
- Se soporta a menudo bajo acoplamiento.
- Se incrementa la reutilización, porque una clase cohesiva se puede usar para un propósito muy específico.

Controlador

Problema

¿Quién debe ser responsable de gestionar un evento de entrada al sistema?

Un *evento del sistema* es un evento generado por un actor externo. Se asocian con *operaciones del sistema* (operaciones del sistema como respuesta a eventos del sistema), de la misma forma que se relacionan los mensajes y los métodos.

Por ejemplo, cuando un escritor utiliza un procesador de textos y presiona el botón "comprobar ortografía", está generando un evento del sistema que indica que se "ejecute una comprobación de la ortografía".

Un *Controlador* es un objeto que no pertenece a la interfaz de usuario. Es responsable de recibir o manejar un evento del sistema y define el método para la operación del sistema.

Solución

Asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones:

- Representa el sistema global, dispositivo o subsistema (*controlador de fachada*).
- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema (*controlador de sesión o de caso de uso*).

  Se debe usar la misma clase controlador para todos los eventos del sistema en el mismo escenario de casos de uso.

Ejemplo (ver transparencia 17)

Discusión

Los sistemas reciben eventos de entrada externos, normalmente a través de diversos medios de entrada. En todos los casos, si se utiliza un diseño de objetos, se debe escoger algún manejador para estos eventos. El patrón Controlador proporciona guías acerca de las opciones generalmente aceptadas y adecuadas. El controlador es una especie de fachada en la capa del dominio para la capa de la interfaz.

A menudo, es conveniente utilizar la misma clase controlador para todos los eventos del sistema de un caso de uso, de manera que sea posible mantener la información acerca del estado del caso de uso en el controlador. Tal información puede ser útil, por ejemplo, para identificar eventos del sistema que se apartan de la secuencia establecida (por ejemplo, una operación de *realizarPago* antes de la operación *finalizarVenta*). Podrían utilizarse diferentes controladores para casos de uso distintos.

Un error típico del diseño de los controladores es otorgarles demasiada responsabilidad.

Normalmente, un controlador debería *delegar* en otros objetos el trabajo que se necesita hacer; coordina o controla la actividad. No realiza mucho trabajo por sí mismo.

Contraindicaciones (consecuencias malas)

Controladores saturados. Una clase controlador pobremente diseñada tendrá una baja cohesión (no centrada en algo concreto y que gestiona muchas áreas de responsabilidad); este controlador se denomina *controlador saturado*. Signos de que existe un controlador saturado son, entre otros:

- Existe una *única* clase controlador que recibe *todos* los eventos del sistema en el sistema y hay muchos.
- El propio controlador realiza muchas de las tareas necesarias para llevar a cabo los eventos del sistema, sin delegar trabajo.

Hay varios remedios para un controlador saturado entre los que se encuentran:

(1)Añadir más controladores, puesto que un sistema no tiene que tener sólo uno

(2) Diseñar el controlador de manera que, ante todo, delegue el cumplimiento de cada responsabilidad de una operación del sistema a otros objetos.

Beneficios (consecuencias buenas)

- Asegura que la lógica de la aplicación *no* se maneja en la capa de interfaz.

- Aumento de la reutilización y bajo nivel de acoplamiento
- Posibilidad de razonar sobre el estado de los casos de uso

Elaboración del modelo de interacción de objetos

En la elaboración del modelo de interacción de objetos, se presta especial atención a la aplicación de los patrones GRASP para desarrollar una solución bien diseñada. Los patrones GRASP, en sí mismos, no son lo importante; simplemente ayudan a discutir y a realizar de manera metódica el diseño de objetos.

En la transparencia 18 se muestran las directrices generales para la elaboración de este modelo

Una realización (o diseño) de caso de uso describe cómo se realiza (diseña) un caso de uso particular en el modelo de diseño, en función de los objetos que colaboran

Los diagramas de interacción UML son un lenguaje común para ilustrar la realización de los casos de uso. Y como se vio en la sección anterior, existen principios o patrones de diseño de objetos, como el Experto en Información, que se pueden aplicar durante este trabajo de diseño.

En la transparencia 19 se muestran los pasos a seguir para el diseño de los casos de uso.

En las transparencias 20 a 32, se ilustra este proceso para una operación concreta.

Tal y como se expresa en los pasos a seguir, el punto de partida para el diseño de una operación es el contrato de esa operación y la parte de diagrama conceptual en el que aparecen los conceptos relacionados con los objetos que intervienen en la interacción.

Elaborar los diagrama de interacción

Considerar el diagrama de conceptos y el contrato de la operación

En la transparencia 20 se muestra el contrato de la operación:

definirProyecto (idProfesor, titulo, numAlum, descrip, listIdAsig)

En la definición de la responsabilidad se han marcado en rojo los conceptos relacionados con los objetos que intervienen de alguna forma en la operación y que hay que considerar en el modelo conceptual. En el ejemplo, esos conceptos son proyecto, profesor, plan de estudios y asignatura

En la transparencia 21 se muestra la parte del modelo conceptual que hay que tener en cuenta en el diseño de la operación.

Representar las relaciones del controlador con los objetos que intervienen en la interacción

La primera decisión de diseño comprende la elección del controlador para el mensaje de operación del sistema. En nuestro caso, como controlador se ha elegido la propia clase que representa al sistema.

A continuación, es necesario responder a la cuestión de ¿qué objetos necesita conocer directamente el controlador? La respuesta es que necesita conocer a todos los profesores, a todos los proyectos y al plan de estudios (ver transparencia 22). No es necesario que conozca directamente las asignaturas, puesto que las conoce el plan de estudios.

El hecho de que el controlador tenga que conocer a todos los profesores y a todos los proyectos se representa en el diagrama de comunicación con un elemento que se denomina multiobjeto. Un multiobjeto es una colección de objetos de un tipo determinado. Los mensajes típicos a los multiobjetos son, entre otros, los de buscar y añadir.

El controlador tiene que conocer también al plan de estudio, pero del plan de estudios lo que tiene es que conocer su estructura, por lo tanto este concepto se representa como una clase.

En la transparencia 22 se puede ver cómo se representan gráficamente los multiobjetos (:Proyecto y :Profesor) y cómo se representa una clase (:PlanEstudios)

Asignar responsabilidades a objetos

El siguiente paso es asignar responsabilidades a los objetos es decir ¿Quién se va a encargar de satisfacer las excepciones y las poscondiciones?.

En el ejemplo, el controlador va a ser responsable de satisfacer las excepciones respecto al profesor y al número de alumnos, También va a ser responsable de la primera poscondición, es decir, de la creación del nuevo proyecto

Para la excepción de comprobar si existe, o no, el profesor identificado por idProfesor, el controlador envía un mensaje de buscar al multiobjeto Profesor, es decir a la colección de objetos del tipo profesor. Si el profesor existe, lo almacena en una variable local (prof), si no existe, el propio multiobjeto lanzará el mensaje de error y la operación se detendrá.

Para comprobar la excepción respecto al número de alumnos, se crea una clase excepción a la que el controlador envía un mensaje de crear ("mensaje de error") si se satisface la condición, o si no se satisface, depende de cómo se exprese dicha condición. En la transparencia 24 y 25 se muestra cómo se envía este mensaje. En este ejemplo, se comprueba que el número de alumnos no sea ni menor de 0 ni mayor de 4

Todos los mensajes que se envíen entre objetos del diagrama, llevan una flecha que indica el sentido del mensaje. Además, los envíos están numerados, puesto que indican el orden en el que se envían dichos mensajes. El envío de un mensaje puede estar constituido por otros envíos. Por ejemplo si el envío del mensaje número 2, requiere enviar otros mensajes, estos mensajes se numeran como 2.1, 2.2,…., y así sucesivamente.

Para satisfacer la primera poscondición de crear un nuevo proyecto, el controlador envía el mensaje de crear al objeto de tipo proyecto, En la transparencia 24 se muestra cómo se representa un objeto particular de una clase concreta (pro:Proyecto)

La creación del nuevo proyecto requiere los argumentos título, número de alumnos, descripción y una lista de asignaturas recomendadas,

¿Quién conoce las asignaturas del plan de estudios? Las asignaturas las conoce el plan de estudios, por lo tanto, el controlador envía a esta clase el mensaje de obtener esa lista de asignaturas a partir del argumento de la operación listIdAsig, Este argumento incluye una lista de identificadores de asignaturas. El plan de estudios buscará esas asignaturas y las almacenará en la variable local listaAsig si existen todas. Si alguna de ellas no existe (lo cual constituye una excepción), se generará un mensaje de error y se detendrá la operación (ver transparencia 24)

Una vez se conocen las asignaturas, se puede proceder a la creación del nuevo proyecto. Una vez creado, es necesario añadir ese nuevo proyecto a la colección de objetos de tipo Proyecto. En consecuencia, el controlador envía al multiobjeto Proyecto el mensaje de añadir el nuevo proyecto creado (ver transparencia 25).

En la transparencia 25 se puede ver la secuencia de numeración de los mensajes.

¿Por qué el mensaje de crear el nuevo proyecto y de añadirlo a la colección de objetos tiene el mismo número de orden (4)?

Porque son dos envíos que se hacen a la vez, es decir, en el mismo momento.

Hay que seguir asignando responsabilidades, puesto que todavía hay que satisfacer dos poscondiciones mas.

La responsabilidad de esas dos poscondiciones (crear enlaces) se le asigna al nuevo proyecto creado (ver transparencia 26). Hay que tener en cuenta que la creación de estos enlaces forma parte de la propia operación de crear el nuevo objeto.

Para crear el enlace entre el nuevo proyecto y el profesor, el proyecto creado envía a la clase Profesor el mensaje de incluir el proyecto *pro* entre los definidos por el profesor *prof*, (el profesor *prof* aparece como argumento de la operación de creación del proyecto).

El enlace entre proyecto y profesor es un doble enlace, en el sentido de que cada proyecto conoce al profesor que lo ha definido y cada profesor conoce los proyectos que él ha definido

Para crear la segunda parte de ese doble enlace (entre el profesor *prof* y el nuevo proyecto *pro*) , el profesor envía un mensaje de incluir el nuevo proyecto al multiobjeto Proyecto, entendiendo este multiobjeto como la colección de proyectos definidos por el profesor *prof*.

La última poscondición de crear enlaces entre el nuevo proyecto y las asignaturas recomendadas, se lleva a cabo con el estéreotipo {new} que crea los enlaces correspondientes entre el proyecto *pro* y todos los objetos contenidos en el multiobjeto Asignatura, entendiendo este multiobjeto como la colección de asignaturas recomendadas para ese proyecto e indicadas como parámetro de la operación de creación

El estereotipo {new} (ver transparencias del seminario de diagramas de interacción) denota la creación de instancias o enlaces durante la interacción.

En la transparencia 27 se muestra el diagrama de comunicación obtenido hasta ahora.

Establecer tipos de enlaces entre objetos: estereotipos de visibilidad

La visibilidad es la capacidad de un objeto de ver o tener una referencia a otro objeto.

En la transparencia 28 se muestra la definición de los cuatro tipos de visibilidad y el estereotipo que le corresponde a cada una de ellas.

En la transparencia 29 se ilustra la forma en que se puede saber el tipo de visibilidad de cada enlace en un diagrama de interacción.

En la transparencia 30, se presenta el ejemplo anterior al que se le han añadido los estereotipos de visibilidad