Introducción a los patrones

Jordi Pradel i Miquel
José Antonio Raya Martos

PID_00165657

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Los **patrones** son una herramienta de popularidad creciente en el desarrollo de software, ya que nos permiten aplicar las ventajas de la reutilización (sobradamente conocidas en el caso del código) a todos los ámbitos del desarrollo (especialmente al análisis y al diseño). Los patrones nos presentan soluciones ya probadas y documentadas que nos pueden ayudar a refinar los artefactos resultantes de las diferentes etapas del ciclo de vida del desarrollo.

Este módulo introduce al estudiante en el mundo de los patrones aplicados a las diferentes etapas del ciclo de vida del desarrollo. Primero veremos qué es un patrón, daremos una descripción formal del mismo, estudiaremos su estructura y veremos las ventajas e inconvenientes de la aplicación de patrones al desarrollo de software mediante un pequeño ejemplo. A continuación haremos una clasificación de los patrones según la etapa del ciclo de vida del desarrollo en que se usan. Veremos algún ejemplo para cada tipo de patrón.

Los ***frameworks*** son otra herramienta muy importante en el desarrollo de software que, al igual que los patrones, nos ayudan a incrementar la calidad del software y, al mismo tiempo, reducir el tiempo de desarrollo. En este módulo hablaremos de los *frameworks*, de su relación con los patrones y de cómo nos pueden ayudar a aplicar patrones al desarrollo de software.

Hay que tener en cuenta que en este módulo se utiliza el lenguaje de modelado UML para la representación de los diagramas tanto del análisis como del diseño.

Objetivos

El objetivo principal de este módulo es introducir el concepto de patrón y entender su aplicación a lo largo del ciclo de vida del desarrollo del software. Por lo tanto, los objetivos que tienen que haber alcanzado los estudiantes cuando acaben este módulo didáctico son los siguientes:

1. Entender el concepto de patrón y saber cómo se documenta un patrón.

2. Comprender las principales ventajas del uso de patrones.

3. Aprender a aplicar los patrones a lo largo del ciclo de vida del desarrollo.

4. Conocer una clasificación de los patrones según la etapa del ciclo de vida en que se pueden aplicar.

5. Entender el concepto de *framework* y su relación con los patrones.

1. Concepto de patrón

1.1. Definición de patrón

Según el *Diccionario de uso del español* de María Moliner un patrón es, entre otras acepciones, una "cosa que se toma como modelo o punto de referencia para medir o valorar otras de la misma especie". También encontramos la expresión *cortados por el mismo patrón* que: "se aplica a cosas y, particularmente, a personas, que se parecen mucho o, sobre todo, que tienen la msima manera de ser (...)".

Gamma, Helm, Johnson y Vlissides [GOF] definen un patrón de la manera siguiente: "un patrón de diseño da nombre, motiva y explica de manera sistemática un diseño general que permite solucionar un problema recurrente de diseño en sistemas orientados a objetos. El patrón describe el problema, la solución, cuándo se tiene que aplicar la solución, así como sus consecuencias. También da algunas pistas sobre cómo implementarlo y ejemplos. La solución es una distribución de objetos y clases que solucionan el problema. La solución se tiene que personalizar e implementar con el fin de solucionar el problema en un contexto determinado".

Así pues, podríamos definir un patrón de la forma siguiente:

> Un patrón es una plantilla que utilizamos para solucionar un problema durante el desarrollo del software con el fin de conseguir que el sistema desarrollado tenga las mismas buenas cualidades que tienen otros sistemas donde anteriormente se ha aplicado la misma solución con éxito.

El hecho de abstraer los elementos generales de una solución concreta para conseguir esta plantilla es lo que nos permite reutilizar esta solución que elaboramos para otro sistema en el sistema que estamos desarrollando.

1.2. Ventajas del uso de patrones

Los patrones no nacen de la inspiración del momento, sino que son el resultado de haber solucionado un mismo problema varias veces y haber comprobado cuáles son las consecuencias de aplicar una solución determinada. De esta manera, los patrones nos permiten aprovechar la experiencia previa a la hora de resolver nuestro problema. Un patrón nos permite lo siguiente:

- Reutilizar las soluciones y aprovechar la experiencia previa de otras personas que han dedicado más esfuerzo a entender los contextos, las soluciones y las consecuencias del que nosotros queremos o podemos dedicar.

- Beneficiarnos del conocimiento y la experiencia de estas personas mediante un enfoque metódico.

- Comunicar y transmitir nuestra experiencia a otras personas (si definimos nuevos patrones).

- Establecer un vocabulario común para mejorar la comunicación.

- Encapsular conocimiento detallado sobre un tipo de problema y sus soluciones asignándole un nombre con la finalidad de poder hacer referencia a éstos fácilmente.

- No tener que reinventar una solución al problema.

1.3. Inconvenientes de un uso inadecuado de patrones

El hecho de disponer de un catálogo de soluciones a problemas generales no evita que tengamos que razonar sobre nuestro desarrollo para entender los problemas que se nos plantean durante el proceso. Así, por ejemplo, durante el diseño, tenemos que saber cuál es nuestro objetivo con el fin de valorar si la aplicación de una solución concreta es mejor que otra. Es bastante habitual que cuando alguien empieza a trabajar con patrones, asuma que el hecho de aplicar un patrón proporciona, automáticamente, la mejor solución posible, lo cual no siempre es cierto.

Una vez escogido un patrón que parezca adecuado (es decir, que sea aplicable al contexto donde nos encontramos), tenemos que asegurarnos de haber entendido correctamente cuál es el problema que quiere solucionar y cuáles son las consecuencias de aplicarlo. Un uso poco cuidadoso del patrón nos puede llevar a una solución final inadecuada, sea porque el patrón no resuelve el problema a que nos enfrentábamos o porque no hemos sabido valorar correctamente las consecuencias de su aplicación y los inconvenientes superan las ventajas.

1.4. Breve perspectiva histórica

El término *patrón* proviene del mundo de la arquitectura y el urbanismo. A finales de los años setenta, Christopher Alexander, junto con otros autores, publican el libro *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)* [AIS]. Este libro contiene unos 250 patrones descritos en un formato específico y estandarizado que describen aquellos principios del diseño, la planificación y la construcción de edificios que, a lo lar-

go de su experiencia, habían detectado que se mantenían sin variaciones en cualquier situación. De esta manera consiguieron transmitir un conocimiento que, hasta entonces, sólo se podía adquirir con años de experiencia.

La mayoría de los patrones de Christopher Alexander siguen la representación siguiente:

```
IF you find yourself in CONTEXT
 for example EXAMPLES,
 with PROBLEM,
 entailing FORCES
THEN for some REASONS,
 apply DESIGN FORM AND/OR RULE
 to construct SOLUTION
 leading to NEW CONTEXT and OTHER PATTERNS
```

Durante los años ochenta muchos autores empezaron a trabajar en este campo buscando la manera de trasladar los beneficios de los patrones al desarrollo de software. A finales de los ochenta y principios de los noventa personas como Ward Cunningham, Kent Beck, Erich Gamma y otros habían empezado a recoger su experiencia en forma de patrones, publicándola con el fin de establecer lenguajes de patrones que funcionaran como los lenguajes de patrones creados por Alexander.

En 1994 se celebró la primera conferencia PLoP[1] dedicada a la discusión y mejora de la publicabilidad de los patrones de diseño. Este mismo año, la Banda de los Cuatro (GOF[2]) finalizó el trabajo que había empezado el año 1990 y publicó el libro *Design Patterns. Elements of Reusable Object-Oriented Software* [GOF], que documentaba, siguiendo un formato estandarizado, un catálogo con los veintitrés patrones de diseño fundamentales que habían encontrado durante estos años. El libro se convirtió rápidamente en un éxito de ventas y muchos arquitectos pudieron reconocer en el catálogo las soluciones a las que habían llegado a ellos mismos, estableciendo por primera vez un lenguaje común adoptado de manera mayoritaria por la comunidad de desarrolladores.

(1)Pattern Languages of Programs http://hillside.net/plop/pastconferences.html

(2)Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides son conocidos popularmente como la Banda de los Cuatro o, en inglés, *Gang of Four*.

1.5. Estructura de un patrón

Los elementos de la estructura de un patrón varían según los autores, si bien hay un conjunto de elementos mínimos comunes a la mayoría de ellos.

Los cinco elementos fundamentales que tiene que tener todo patrón son el nombre, el contexto, el problema, la solución y las consecuencias de aplicarlo.

El **nombre** nos permite hacer referencia al patrón cuando documentamos nuestro diseño o cuando lo comentamos con otros desarrolladores. También nos permite aumentar el nivel de abstracción del proceso de diseño, ya que podemos pensar en la solución al problema como un todo.

El **contexto** nos indica qué condiciones se tienen que dar para que el patrón sea aplicable.

El **problema** nos indica qué resuelve el patrón. Este problema podría ser un problema concreto o la identificación de una estructura problemática (por ejemplo, una estructura de clases poco flexible).

La **solución** describe los elementos que forman parte del patrón, así como sus relaciones, responsabilidades y colaboraciones. La solución no es una estructura concreta, sino que, como hemos indicado anteriormente, es como una plantilla que podemos aplicar a nuestro problema. Algunos patrones presentan varias variantes de una misma solución.

Las **consecuencias** son los resultados y los compromisos derivados de la aplicación del patrón. Es muy importante que las tengamos en cuenta a la hora de evaluar nuestro diseño, ya que es lo que nos permite entender realmente el coste y los beneficios de la aplicación del patrón.

1.6. Aplicación de un patrón

¿Cómo sabemos cuándo tenemos que aplicar un patrón? Seguiremos el proceso siguiente:

1) Identificar el problema que queremos resolver. Lo primero que hay que tener claro es cuál es el problema que estamos intentando resolver. Es posible que tengamos que descomponer un problema complejo en diferentes subproblemas para tratarlos de manera individual.

2) Plantear posibles soluciones al problema. Estas soluciones las plantearemos de manera independiente al catálogo de patrones disponible. El hecho de plantear las soluciones antes de consultar el catálogo nos permitirá tener un conocimiento más profundo del problema en el momento de seleccionar los patrones del catálogo.

3) Identificar aquellos patrones que resuelven el **problema** que hemos identificado. Habrá que acudir a nuestro catálogo de patrones buscando qué patrones podrían resolver el problema que nos hemos planteado.

4) Descartar aquellos patrones cuyo **contexto** no sea compatible con el contexto en que nos encontramos. Para que la aplicación de un patrón sea beneficiosa, nos tenemos que asegurar de que no sólo soluciona el problema que nos interesa, sino que, además, lo hace en un contexto compatible con el nuestro.

Por ejemplo, no nos sirve de nada un patrón arquitectónico que nos ayuda a estructurar una aplicación cuando la arquitectura es centralizada si nuestra arquitectura es distribuida.

5) Valorar las **consecuencias** de cada solución. De cada una de las soluciones que hemos preseleccionado, nos tenemos que plantear las consecuencias de su aplicación. Con respecto a las soluciones basadas en patrones, tenemos la ventaja de que las consecuencias generales han sido estudiadas y documentadas previamente formando parte de la descripción del patrón. Para el resto de soluciones, tendremos que estudiar las consecuencias con el riesgo de hacer un estudio incompleto y sin la ayuda de experiencia previa. Finalmente, si un patrón tiene varias variantes, tenemos que valorar las consecuencias de cada una de éstas.

6) Escoger y aplicar una solución. Una vez valoradas las consecuencias, ya disponemos de un criterio bien fundamentado para escoger la solución más adecuada y, por lo tanto, podemos proceder a su aplicación.

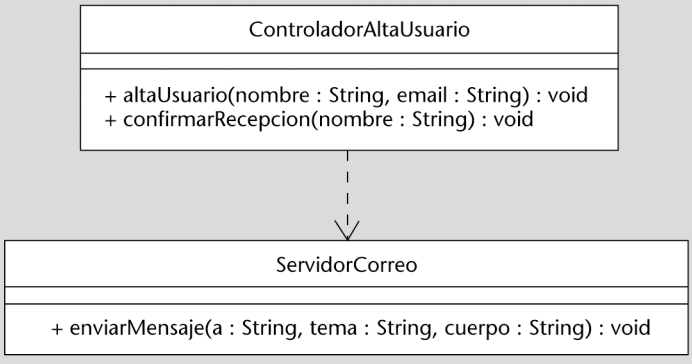
Como se puede ver, el proceso de aplicación de un patrón no se diferencia sustancialmente de cualquier proceso de toma de decisiones ante un problema. En el caso de los patrones, sin embargo, nos ahorramos el estudio de las variantes, de sus consecuencias, y tenemos una guía a la hora de elaborar una solución al problema.

1.6.1. Ejemplo

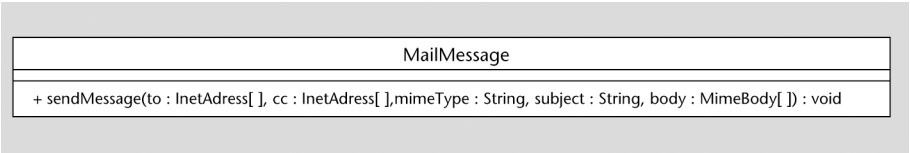
Supongamos que, diseñando nuestro sistema, nos encontramos con un requisito según el cual durante el alta de un usuario hay que enviar una notificación por correo electrónico a la dirección que el usuario indique. Aplicando el principio de inversión de dependencias, empezamos a diseñar esta clase identificando las operaciones que necesitamos:

Ved también

Podéis ver el principio de inversión de dependencias en el apartado 2.7 del módulo 2, "Catálogo de patrones".



Este problema ya lo tuvimos que resolver en el pasado en otro proyecto y nos planteamos que quizás podemos reutilizar la solución que obtuvimos:



Nos encontramos, sin embargo, con un problema: las operaciones que queremos utilizar no son exactamente las mismas que ofrece la clase que queremos reutilizar.

Ante este problema se nos ocurren dos soluciones alternativas:

a) Implementar la clase ControladorAltaUsuario utilizando las operaciones que le ofrece la clase MailMessage, de tal manera que el controlador dependa directamente de MailMessage.

b) Modificar la clase MailMessage para añadir las operaciones de ServidorCorreo. De esta manera, se reutiliza la clase, pero cambiando su implementación.

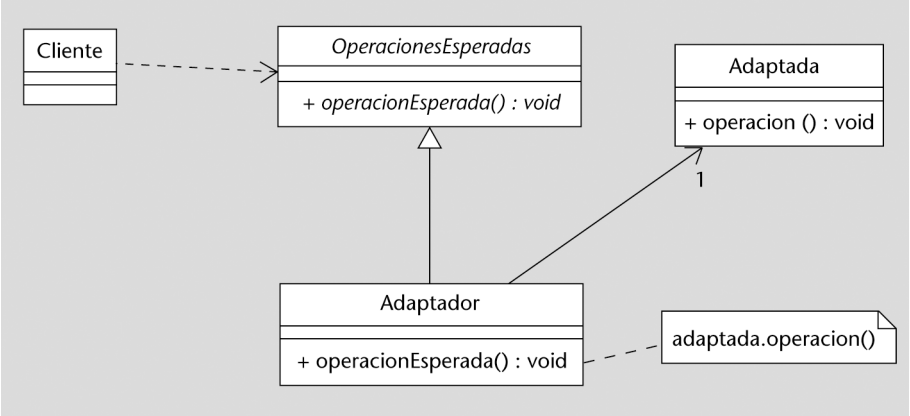
¿Hay alguna otra manera de solucionar este problema? Consultamos el catálogo y encontramos un patrón que resuelve nuestro problema:

- Nombre: Adaptador

- Contexto: existe una clase que nos proporciona una funcionalidad que queremos aprovechar.

- Problema: las operaciones que nos ofrece la clase que queremos reutilizar no son exactamente las que queremos utilizar.

- Solución: añadir una clase Adaptador que ofrezca las operaciones que espera la clase Cliente, pero que delegue la implementación a la clase que queremos reutilizar:



- Consecuencias:
  - Permite al Adaptador redefinir parte del comportamiento de Adaptada.
  - Un mismo Adaptador puede funcionar con diferentes subclases de la clase Adaptada. El Adaptador también puede añadir funcionalidad a toda la jerarquía de herencia al mismo tiempo.
  - Podemos sustituir la clase Adaptada por otra implementación desarrollando un nuevo Adaptador.

Una vez hemos visto que hay un patrón que parece, *a priori*, aplicable a nuestro problema, lo primero que tenemos que hacer es verificar que el contexto del patrón se adecua a nuestra situación. ¿Estamos ante la situación en la que "existe una clase que nos proporciona una funcionalidad que queremos aprovechar"? En este caso, es bastante claro que sí.

El paso siguiente será estudiar las consecuencias de las tres soluciones que hemos encontrado:

a) Modificar la clase ControladorAltaUsuario.

- Añadimos una dependencia hacia la clase MailMessage de manera que si la queremos sustituir por otra implementación, tendremos que volver a modificar ControladorAltaUsuario.
- Como no estamos aplicando un patrón, no tenemos experiencia que nos pueda garantizar que esta solución no tenga consecuencias de las cuales en este momento no somos conscientes.

b) Modificar la clase MailMessage.

- Perdemos la capacidad de reutilización de la clase, ya que no compartimos el código entre los dos sistemas donde hemos utilizado MailMessage. Por lo tanto, en lo sucesivo, tendremos que mantener dos versiones diferentes

de la clase MailMessage y las mejoras que introduzcamos en un sistema no se podrán trasladar automáticamente al otro.

- Como en el caso anterior, podría haber consecuencias que no hayamos tenido en cuenta.

c) Adaptador.

- La solución es más compleja, ya que introduce una interfaz y una clase nueva.

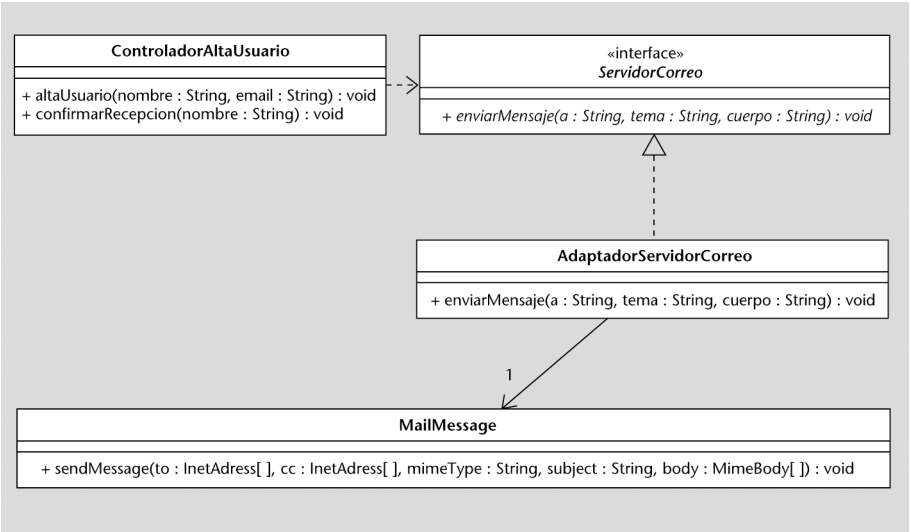
- Las consecuencias ya documentadas del patrón: reutilizamos la implementación que ya tenemos sin cerrarnos la posibilidad de sustituirla por otra en el futuro. Podemos redefinir la funcionalidad de la clase con el fin de conseguir que nos ofrezca exactamente lo que queríamos.

Finalmente, tenemos que escoger una de las soluciones. ¿Tenemos que aplicar el patrón? Depende. Para escoger la solución más adecuada nos tenemos que hacer preguntas como:

a) ¿Es preciso que nuestro sistema sea independiente de la clase que reutilizaremos? ¿Hay muchas clases que tengan la necesidad de enviar mensajes de correo por medio de esta clase? Si reutilizamos MailMessage directamente modificando ControladorAltaUsuario para que utilice las operaciones actuales, un cambio en estas operaciones (que podría ser provocado por la otra aplicación) afectará a nuestro sistema y, posiblemente, habrá que modificarlo.

b) ¿Tenemos que mantener la clase reutilizada independiente de nuestro sistema? Si añadimos nuestras operaciones a MailMessage, el mantenimiento de esta clase será más complicado, ya que tendremos que mantener dos versiones de una misma clase ligeramente diferentes.

La solución que escogeremos dependerá de las respuestas a estas preguntas. Supondremos que queremos mantener nuestro sistema y la clase MailMessage independientes entre sí. En este caso, aplicaremos el patrón Adaptador con el resultado siguiente:



La clase cliente del patrón es ControladorAltaUsuario, que es quien utilizará la clase adaptada MailMessage. Para adaptar esta clase a la interfaz esperada (ServidorCorreo), hemos creado el adaptador AdaptadorServidorCorreo; en este caso, hemos modificado ligeramente la estructura del patrón original, ya que el adaptador implementa una interfaz en lugar de extender una clase.

1.7. Influencia del patrón en el ciclo de vida

La primera aplicación de los patrones que se hizo en el campo de la ingeniería del software fue durante la etapa de diseño. Eran, pues, patrones que documentaban soluciones de diseño a problemas de diseño; de hecho, aún hoy en día, ésta es la etapa del ciclo de vida donde es más común el uso de patrones y donde encontraremos el mayor número de patrones.

Los patrones son aplicables a cualquier tarea donde se puedan identificar soluciones genéricas a un cierto problema. En concreto, a todas las etapas del ciclo de vida de la ingeniería del software se puede aplicar el concepto.

Ejemplo

Recordemos que el ámbito original de los patrones no era ni siquiera la ingeniería del software.

Algunos de los casos en los que se han aplicado los patrones con éxito son los siguientes:

- **Modelos de análisis.** Durante la etapa de análisis nos encontramos con problemas consistentes en decidir qué modelo representa mejor los conceptos y restricciones del mundo real que estamos modelando. Hay muchos puntos en los que nos enfrentamos a problemas de modelización que ya han sido resueltos con anterioridad y los llamados *patrones de análisis* nos pueden ayudar. Algunos ejemplos pueden ser la representación de las asociaciones de las cuales queremos conocer información histórica o la modelización de sistemas que trabajen con reglas de contabilidad.

- **Arquitectura de software**. A modo de primeros pasos en la etapa de diseño (o como etapa diferenciada según el ciclo de vida que se siga) habrá que definir la arquitectura global del software que se desarrollará. En este contexto también habrá que solucionar ciertos problemas con posibles soluciones genéricas y se han documentado varios patrones que llamamos *arquitectónicos* que nos pueden ser útiles. Por ejemplo, el patrón de arquitectura en capas nos propone una manera de organizar internamente nuestra arquitectura para minimizar la complejidad del sistema resultante.

- **Asignación de responsabilidades**. Durante el diseño (sobre todo en sus fases iniciales) tenemos que llevar a cabo la asignación de responsabilidades a las clases que forman el sistema. Por ejemplo, el patrón Creador nos da un criterio general para decidir qué clase tendrá la responsabilidad de crear instancias de otra clase.

- **Diseño de software.** Durante la etapa de diseño es cuando los patrones son más utilizados. Hay un amplio abanico de *patrones de diseño* documentados que nos permiten solucionar problemas de diseño frecuentes con soluciones comunes ya conocidas. Por ejemplo, tal como hemos visto anteriormente, el patrón Adaptador nos da una solución al problema concreto de reutilizar una clase sin utilizar exactamente la interfaz que ofrece.

- **Programación.** Una vez fijada una tecnología y un lenguaje de programación, encontraremos problemas que requieren soluciones que no pueden ser genéricas al diseño o a etapas anteriores, sino que han sido originadas por el lenguaje de programación escogido y, por lo tanto, dependen de él. Los llamados *patrones de programación*[3] son, pues, patrones que nos dan soluciones específicas a un lenguaje de programación. Como ejemplo, hay patrones que nos guían en el uso de interfaces en Java.

(3)En inglés, *idiom*.

2. Tipos de patrones

Hay muchas posibles taxonomías de los patrones de la ingeniería del software, cada una según criterios diferentes. Nosotros tipificaremos los patrones dependiendo de la etapa del ciclo de vida en que se utilizan, ya que es una de las tipificaciones más aceptadas y, al mismo tiempo, útiles.

Éste no se considerará, sin embargo, un criterio estricto, ya que la frontera entre una etapa y otra no siempre es clara. Así, por ejemplo, algunos patrones que nosotros clasificaremos como de arquitectura son considerados por muchos autores patrones de diseño.

Distinguiremos los tipos de patrones siguientes:

- Patrones de análisis
- Patrones arquitectónicos
- Patrones de asignación de responsabilidades
- Patrones de diseño
- Patrones de programación

En esta asignatura haremos hincapié en los patrones de asignación de responsabilidades y de diseño, pero también veremos algunos patrones de análisis y de arquitectura.

2.1. Patrones de análisis

Los patrones de análisis son aquellos patrones que documentan soluciones aplicables durante la realización del diagrama estático de análisis para resolver los problemas que surgen: nos proporcionan maneras probadas de representar conceptos generales del mundo real en un diagrama estático del análisis.

Lo podemos ver más claro con un ejemplo: supongamos que estamos desarrollando un sistema de gestión de una estación meteorológica. En el diagrama estático de análisis tenemos una clase Termómetro que tendrá que tener la información sobre la temperatura medida. Podemos pensar en utilizar un atributo de tipo entero que represente este dato, pero nos faltaría una información muy importante para interpretar el valor de este atributo: ¿en qué escala está representada esta temperatura? ¿Qué quiere decir una temperatura de 100? ¿Son grados centígrados, Fahrenheit, Kelvin? Así pues vemos que el atributo entero es claramente insuficiente.

El patrón de análisis Cantidad [FOW] nos permite solucionar este tipo de problema: tenemos que añadir una clase Temperatura que tenga la responsabilidad de saber los grados y la escala en que están.

2.2. Patrones arquitectónicos

Los patrones arquitectónicos son aquellos que se aplican en la definición de la arquitectura de software y que, por lo tanto, resuelven problemas que afectarán al conjunto del diseño del sistema.

Por ejemplo, en la definición inicial de la arquitectura de nuestro sistema nos encontramos con el problema de que su complejidad nos obliga a trabajar a diferentes niveles de abstracción. Queremos que los componentes de presentación no tengan que interactuar directamente con los servicios técnicos a un nivel bajo como la base de datos.

El patrón arquitectónico en capas soluciona este problema proponiendo que cada componente se asigne a una capa con un cierto nivel de abstracción: presentación, dominio y servicios técnicos, de tal manera que cada capa sólo depende de la siguiente. Así, los componentes de presentación trabajarán a su propio nivel de abstracción y delegarán cualquier otra tarea en la capa de dominio.

2.3. Patrones de asignación de responsabilidades

Los patrones de asignación de responsabilidades se utilizan durante la etapa de diseño. Pero este tipo de patrones resuelve problemas durante una tarea muy concreta del diseño orientado a objetos: la asignación de responsabilidades a las clases software.

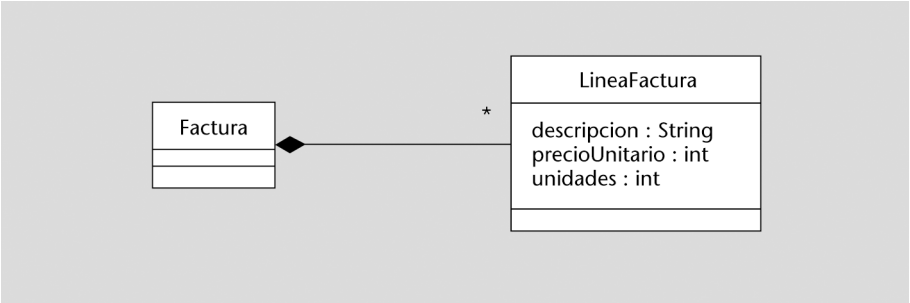
Este tipo de patrones de diseño se diferencia claramente de los patrones de diseño generales en que los problemas que resuelven no son específicos para una situación o sistema concreto, sino que son problemas comunes a cualquier asignación de responsabilidades. Por lo tanto, estos patrones, en realidad, consisten en pautas que nos ayuden a hacer una asignación de responsabilidades siguiendo los principios de diseño más básicos.

Todos los patrones de asignación de responsabilidades comparten una estructura muy parecida. Responden a la pregunta: "¿Qué clase tendrá que tener la responsabilidad R para que el diseño siga unos principios de diseño sólidos?". Cuando iniciamos el diseño, partiendo del modelo conceptual del análisis, creamos un modelo de clases del diseño. En este momento, empezamos a asignar responsabilidades que colaborarán para implementar el comportamiento del sistema.

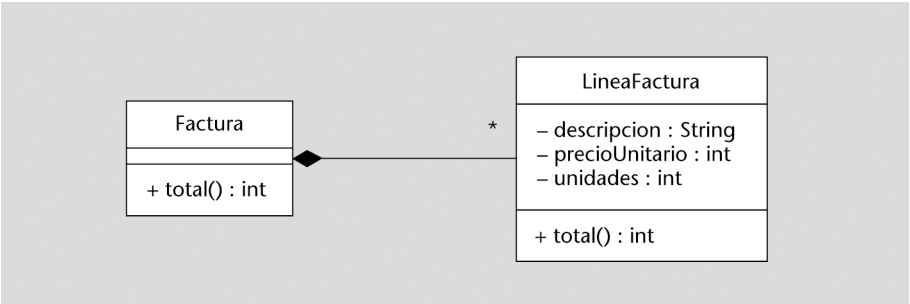
Ejemplo

Larman, autor de la familia de patrones GRASP [LAR], que es el ejemplo más importante de patrones de asignación de responsabilidades, documenta los principios de diseño en forma de patrones. Así, por ejemplo, para Larman existen los patrones Alta cohesión y Bajo acoplamiento.

Supongamos que tenemos el modelo parcial siguiente:



¿Qué clase tendrá la responsabilidad de calcular el total de una factura? Según el patrón de asignación de responsabilidades Experto, será la clase que tenga la información necesaria para hacer este cálculo. En nuestro ejemplo, la clase Factura tiene la información de las líneas de factura que forman una factura y será la experta escogida. Para calcular el total de la factura, sin embargo, esta experta tendrá que sumar el total de cada línea. ¿Qué clase tendrá la responsabilidad de calcular este total de línea? De nuevo, según el patrón Experto, la clase que tiene la información necesaria es LineaFactura. Por lo tanto, nuestro modelo del diseño queda de la manera siguiente:



2.4. Patrones de diseño

Los patrones de diseño son aquellos que se aplican para resolver problemas concretos de diseño que no afectarán al conjunto de la arquitectura del sistema.

No clasificaremos en este grupo los patrones de asignación de responsabilidad, aunque también se utilizan en la etapa de diseño.

El ejemplo que hemos visto anteriormente de aplicación del patrón de diseño Adaptador es un buen ejemplo de aplicación de un patrón de este grupo.

Ved también

Podéis ver el ejemplo del patrón de diseño Adaptador en el apartado 1.6.1 de este módulo.

2.5. Patrones de programación

Un patrón de programación es un patrón que describe cómo se pueden implementar ciertas tareas utilizando un lenguaje de programación concreto.

Por lo tanto, los patrones de programación resuelven problemas que o bien son específicos de un lenguaje de programación o bien son genéricos, pero tienen una buena solución conocida que es específica de un lenguaje de programación.

Por ejemplo, durante el análisis y diseño hemos utilizado en nuestros modelos una enumeración llamada Colores que puede tomar los valores Rojo, Azul y Verde. Cuando implementamos el sistema en Java (versión 1.4) nos encontramos con el problema de que no tiene soporte para enumeraciones. El patrón de programación "Enumeraciones en Java" nos permite resolver este problema creando una clase Color con constructor protegido que tendrá tres atributos estáticos, uno para cada valor posible:

```
public class Color {
 public static final Color ROJO = new Color();
 public static final Color VERDE = new Color();
 public static final Color AZUL = new Color();
 protected Color() {}
}
```

Ahora podremos utilizar esta clase de la manera siguiente:

```
if(color == Color.ROJO)
...
```

En la asignatura nos centraremos en los patrones de más alto nivel y, por lo tanto, no veremos los patrones de programación.

3. *Frameworks*

Un *framework* es un conjunto de clases predefinidas que tenemos que especializar y/o instanciar para implementar una aplicación o subsistema ahorrando tiempo y errores. El *framework* nos ofrece una funcionalidad genérica ya implementada y probada que nos permite centrarnos en lo específico de nuestra aplicación.

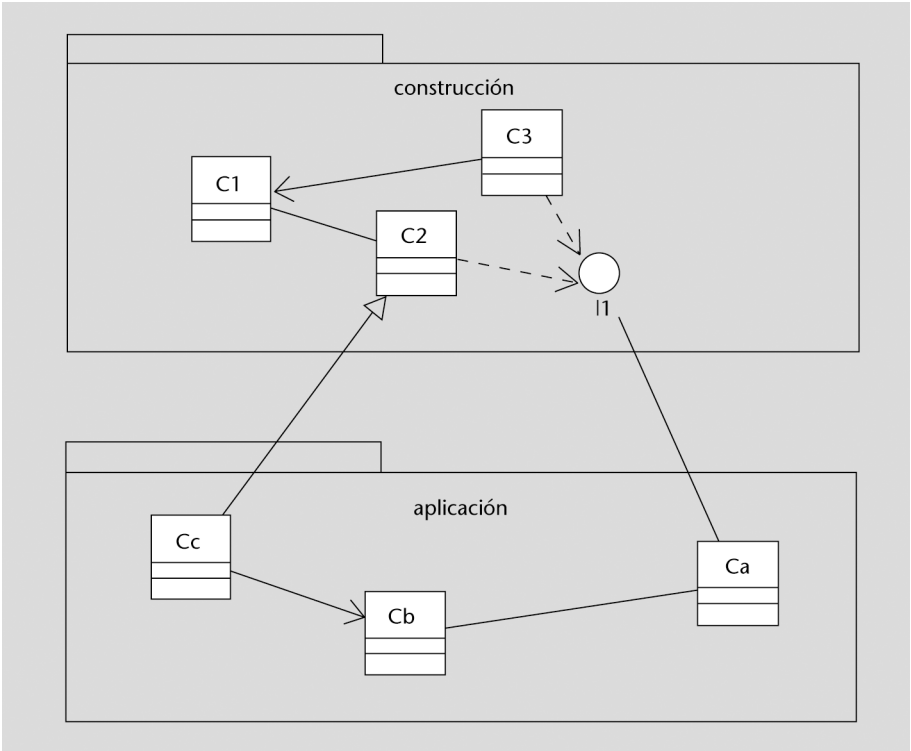
No debemos confundir un *framework* con una biblioteca[4] de clases. Un *framework* es un tipo especial de biblioteca que define el diseño de la aplicación que lo utilizará.

(4)En inglés, *library*.

Un *framework* comparte el control de la ejecución con las clases que lo utilizan siguiendo el "principio de Hollywood[5]": el *framework* implementará el comportamiento genérico del sistema y llamará a nuestras clases en aquellos puntos donde haya que definir un comportamiento específico.

(5)En inglés, "*Don't call us. We'll call you*" (no hace falta que nos llames, nosotros te llamaremos).

Como un *framework* tiene que llamar a nuestras clases, definirá la interfaz que espera que tengan estas clases. Así, para utilizar un *framework* tendremos que implementar el comportamiento específico de nuestra aplicación respetando las interfaces y colaboraciones definidas en él.



En este caso, por ejemplo, nuestra aplicación utiliza un *framework* mediante la herencia (de la clase Cc hacia C2) y la implementación de una interfaz (Ca implementa I1) que es utilizada por el *framework* (por C2 y C3). De esta manera, el *framework* llevará el control de la ejecución llamando a las clases Ca y Cc de nuestra aplicación cuando convenga.

3.1. Ventajas e inconvenientes del uso de *frameworks*

Cuando utilizamos un *framework*, reutilizamos un diseño ahorrando tiempo y esfuerzo. Por otra parte, dado que el control es compartido entre la aplicación y el propio *framework*, un *framework* puede contener mucha más funcionalidad que una biblioteca tradicional.

Otra ventaja del uso de *frameworks* es que nos dan una implementación parcial del diseño que queremos reutilizar.

El principal inconveniente del uso de *framework* es la necesidad de un proceso inicial de aprendizaje. Para reutilizar el diseño del *framework*, primero lo tenemos que entender completamente.

3.2. Relación entre *frameworks* y patrones

La principal diferencia entre un patrón y un *framework* es que el *framework* es un elemento de software, mientras que el patrón es un elemento de conocimiento sobre el software. Por lo tanto, la aplicación de un patrón partirá de una idea, pero no de una implementación; de hecho, el patrón se tendrá que implementar desde cero adaptándolo al caso concreto. En cambio, la aplicación de un *framework* partirá de un conjunto de clases ya implementadas cuya funcionalidad extenderemos sin modificarlas.

Desde el punto de vista de los patrones, un *framework* puede implementar para nosotros un conjunto de patrones. Es decir, que el diseño que reutilizamos puede estar basado en patrones reconocidos igual que si lo hubiéramos hecho nosotros. En este caso, entender los patrones en los cuales se basa un *framework* nos ayudará a entender su funcionamiento y a poder utilizarlo eficazmente.

3.3. Uso de *frameworks*

El uso de *frameworks* afecta a nuestro proceso de desarrollo de software. Dado que estamos reutilizando un diseño previo, tenemos que acomodar nuestro sistema al diseño que queremos reutilizar.

Aunque eso pueda parecer una limitación, la capacidad de plantear nuestro diseño en términos de un diseño aplicado previamente con éxito por los creadores del *framework* nos ayudará a garantizar la calidad de este diseño.

3.4. Un ejemplo de *framework*: Hibernate

Cuando desarrollamos un sistema que almacena sus datos en una base de datos relacional, es habitual encontrarse con el problema de sincronizar los datos de los objetos que tenemos en memoria con los datos de la base de datos relacional. Este problema es la causa de que muchas aplicaciones informáticas no puedan sacar todo el provecho de la tecnología orientada a objetos, ya que tienen que trabajar directamente con el modelo de datos relacional.

Hibernate es un *framework* de persistencia de objetos Java a bases de datos relacionales que nos permite añadir, de manera transparente, la capacidad de persistencia a las clases que diseñemos. Esta estructura nos ofrece una serie de mecanismos por los cuales podemos trabajar con objetos persistentes de la base de datos desde un punto de vista orientado a objetos con soporte, por lo tanto, de conceptos como asociaciones, herencia, polimorfismo, composición y uso de colecciones. Hibernate también define un lenguaje de consultas propio denominado *Hibernate query language*, que nos permite hacer consultas con la capacidad de SQL con algunas extensiones para soportar el uso de objetos.

Para desarrollar una aplicación con Hibernate es imprescindible haber hecho el modelo relacional y el diseño de las clases. Partiendo de este punto definiremos la correspondencia entre clases Java y tablas del modelo relacionales mediante un lenguaje basado en XML. Entonces sólo hay que configurar cómo se conectará a la base de datos a fin de que Hibernate se ocupe de manera transparente de sincronizar el modelo de datos orientado a objetos en memoria y el modelo relacional del servidor de base de datos.

3.4.1. Ventajas

- Permite trabajar sobre bases de datos relacionales utilizando toda la potencia de la orientación a objetos. Esta característica propicia reducir la complejidad del desarrollo con el consiguiente aumento de productividad, fiabilidad y mantenibilidad.

- Incluye numerosas mejoras de rendimiento respecto de otras soluciones como la carga de datos bajo petición (que evita cargar datos que finalmente no se consulten), la optimización transparente de consultas SQL (gracias al conocimiento que tiene el *framework* de los diferentes SGBDR[6]) o el uso de una memoria caché[7].

- Permite cambiar de dialecto SQL (y, por lo tanto, de fabricante de bases de datos) sin tener que modificar el código.

[6] Sistema Gestor de Bases de Datos Relacionales.

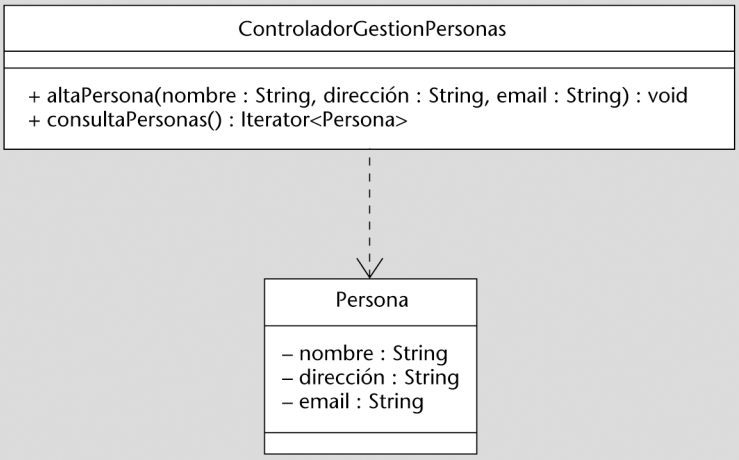
[7] en inglés, *cache*.

3.4.2. Inconvenientes

El principal inconveniente de Hibernate es la necesidad de un proceso inicial de aprendizaje tanto con respecto a su modelo de programación como a la interfaz específica que nos ofrece.

3.4.3. Ejemplo de uso

Queremos diseñar una aplicación de gestión de datos sobre personas con el siguiente diseño de clases del dominio:



Nuestro modelo relacional es muy sencillo:

```
create table Persona (
  nombre varchar (32),
  dirección varchar (80) not null,
  email varchar(80) not null,
  primary key (nombre)
)
```

Para adaptar nuestro diseño a Hibernate tenemos que definir la correspondencia entre clases i tablas mediante el documento XML siguiente:

```
<hibernate-mapping>
   <class name="dominio.Persona" table="Persona">
   <id name="nombre" type="string" >
     <generator class="assigned"/>
   </id>
<property name="dirección"/>
<property name="email"/>
</class>
```

```
</hibernate-mapping>
```

A continuación habrá que modificar el comportamiento del controlador a fin de que utilice el *framework*:

```
public class ControladorGestionPersonas {
 public void altaPersona(String nombre, String
 dirección, String email) {
     Session session = HibernateUtil.currentSession();
     Transaction tx = session.beginTransaction();
     Persona p = new Persona();
     p.setDireccion(direccion);
     p.setEmail(email);
     p.setNombre(nombre);
     session.save(p);
     tx.commit();
     HibernateUtil.closeSession();
}
public Iterator<Persona> consultaPersonas() {
 List<Persona> resultado = new LinkedList<Persona>;
 Session session = HibernateUtil.currentSession();
 Transaction tx = session.beginTransaction();
 Query query = session.createQuery("from Persona");
 for (Iterator it = query.iterate(); it.hasNext();) {
     Persona p = (Persona) it.next();
     resultado.add(p);
 }
 return resultado.iterator();
}
```

Como se puede ver, la clase Persona no hará falta modificarla, ya que Hibernate se encarga de la persistencia de manera transparente llamando a las operaciones de Persona que haga falta. Con respecto al controlador, aunque tiene una parte de código específica para la persistencia, la proporción de esta parte con respecto al conjunto del código es muy pequeña.

Como hemos visto, el uso de Hibernate es mucho menos intrusivo que el uso, por ejemplo, de sentencias SQL dentro del código. Por otra parte, nos permite trabajar en un nivel de abstracción mucho más alto y nos evita tener que estar pendientes de liberar recursos una vez utilizados, obtener conexiones, etc.

3.4.4. Hibernate y patrones

Hay que destacar que el propio Hibernate utiliza patrones de diseño en su construcción, como, por ejemplo, el patrón Representante. Además, la comunidad de usuarios de esta estructura ha definido todo un catálogo de patrones de programación específicos para Hibernate.

Así pues, queda patente que la relación entre un *framework* y los patrones es doble: por una parte, ayudan a que el *framework* esté mejor diseñado y, por la otra, el uso del *framework* genera un nuevo catálogo de patrones propio.

Resumen

Los patrones se han convertido en una herramienta muy importante en el desarrollo de software porque nos permiten obtener las ventajas de la reutilización en contextos diferentes a la reutilización de código.

Se ha estudiado el concepto general de patrón, cómo se documenta un patrón y cómo se puede aplicar esta herramienta a las diferentes etapas del ciclo de vida del desarrollo. Hemos visto las ventajas y los peligros derivados de un mal uso de los mismos.

También se ha visto una clasificación de patrones según la etapa del ciclo de vida del desarrollo en el que son aplicables. Esta clasificación nos será útil para localizar rápidamente y relacionar entre sí los diferentes patrones del catálogo que se recogerá en el módulo correspondiente.

Finalmente, hemos visto el concepto de *framework* y su relación con el análisis y diseño orientados a objetos con patrones, así como un ejemplo práctico de utilización del *framework* Hibernate.

Ejercicios de autoevaluación

1) Enumerad las ventajas del uso de patrones.

2) Indicad dos causas por las cuales el uso de un patrón se puede considerar inadecuado.

3) ¿El uso de patrones es exclusivo de la ingeniería del software?

4) ¿Cuáles son los elementos fundamentales del *framework* de un patrón?

5) ¿Qué aporta el uso de patrones al proceso de toma de decisiones durante, por ejemplo, el diseño de software?

6) ¿Qué diferencia hay entre un patrón de análisis y un patrón de diseño?

7) Explicad en qué consiste el llamado "principio de Hollywood".

8) Indicad una ventaja y un inconveniente del uso de *frameworks*.

Solucionario

1) Reutilizar soluciones previas, beneficiarnos del conocimiento de otros, comunicar y transmitir la experiencia, establecer un vocabulario común, encapsular conocimiento y no tener que reinventar una solución a un problema.

2) Porque no resuelva el problema al cual nos enfrentábamos o porque no hemos valorado correctamente las consecuencias de su utilización y los inconvenientes superan a las ventajas.

3) No; de hecho, el término *patrón* proviene del mundo de la arquitectura y el urbanismo.

4) Nombre, contexto, problema, solución y consecuencias de aplicación.

5) El proceso viene a ser el mismo, pero nos ahorramos el estudio de las consecuencias de la aplicación del patrón y tenemos una guía a la hora de elaborar una solución para el problema.

6) La diferencia fundamental reside en la etapa del ciclo de vida del desarrollo donde se utilizan uno y otro.

7) Este principio hace referencia al hecho de que el control de ejecución del sistema es compartido entre un *framework* y la aplicación que lo utiliza. Consiste en el hecho de que el *framework* es quien llama a nuestras clases en aquellos puntos donde haya que definir un comportamiento específico.

8) Una ventaja es que nos dan una implementación parcial del diseño que queremos reutilizar. Un inconveniente es que, para reutilizar este diseño, primero lo tenemos que entender completamente y, por lo tanto, es necesario un periodo inicial de aprendizaje.

Glosario

adaptador *m* Patrón de diseño que nos permite reutilizar una clase previamente existente empleando un conjunto de operaciones diferente al que la clase proporciona.

arquitectura en capas *f* Patrón arquitectónico que nos propone organizar nuestra arquitectura en un conjunto de capas que trabajan a diferentes niveles de abstracción.

asignación de responsabilidades *f* Tarea de diseño en la que se decide qué clase se tiene que encargar de cada responsabilidad dentro de un sistema orientado a objetos.

biblioteca de clases *f* Conjunto de clases reutilizables y relacionadas entre sí.

cantidad *f* Patrón de análisis que permite modelar una cantidad indicando la escala en la que está medida.

ciclo de vida *m* Conjunto de etapas del desarrollo de software por las cuales se pasa en un orden establecido previamente.

consecuencias de aplicación de un patrón *f pl* Elemento de la descripción de un patrón que indica los resultados y compromisos derivados de la aplicación del patrón.

contexto *m* Elemento de la descripción de un patrón que indica qué condiciones se tienen que dar para que el patrón sea aplicable.

diagrama estático de análisis *m* Diagrama de clases que representa los conceptos del mundo real que nuestro sistema conoce y manipula.
sin. **modelo conceptual**

experto *m* Patrón de asignación de responsabilidades que propone asignar una responsabilidad a la clase que tenga la información necesaria para llevarla a cabo.

framework *m* Conjunto de clases que tenemos que especializar y/o instanciar para implementar una aplicación o subsistema.

GRASP *m* Principal familia de patrones de asignación de responsabilidades documentada por Craig Larman.

Hibernate *m Framework* de persistencia de objetos Java en bases de datos relacionales.

modelo conceptual *m* sin. **diagrama estático de análisis**

patrón de programación *m* Patrón específico para una tecnología concreta, típicamente un lenguaje de programación.

patrón *m* Plantilla que utilizamos para solucionar un problema durante el desarrollo de software.

patrón de programación *m* sin. **modismo**

Bibliografía

[AIS] Alexander, C. y otros (1977). *A Pattern Language: Towns, Buildings, Construction*. Nueva York: Oxford University Press.

[GOF] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts: Addison Wesley Professional.

[FOW] Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Massachusetts: Addison Wesley Professional.

[LAR] Larman, C. (2003). *UML Y PATRONES. Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Madrid: Prentice Hall.

[POSA] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. (1996). *Pattern-Oriented Software Architecture: A System Of Patterns*. West Sussex, Inglaterra: John Wiley & Sons Ltd.