
Arquitecturas Reflexivas y Generación de Código.

- 1.- Empezemos con un ejemplo: AnotherAmazon.com .
 - 2.- Generación de código fuente (o desarrollo manual en su defecto)..
 - 3.- Qué es una arquitectura reflexiva.
 - 4.- Cuándo desarrollar una arquitectura reflexiva.
-

1.- Empezemos con un ejemplo: AnotherAmazon.com.

Imaginemos que recibimos el encargo de construir un portal comercial orientado a la venta por internet de artículos de consumo como libros, películas, DVDs, CDs, impresoras, ratones, cámaras fotográficas, reproductores mp3, televisores, y toda clase de consumibles electrónicos.

Cada uno de estos tipos de objetos se definen por sus características. De hecho tendremos:

- Características que aparecerán en todos los objetos, como por ejemplo el precio que el comprador debe pagar para poder obtenerlo, así como un peso y volumen, elementos a tener en cuenta en cuanto al transporte, etc.
- Un conjunto de atributos particulares para cada tipo de objeto. Por ejemplo en las cámaras digitales fotográficas la resolución máxima de definición (medida en megapíxeles), o por ejemplo en los CDs y DVDs la duración de las películas o discos de música que contengan.

Para la gestión de toda esta información, surge de forma evidente la necesidad de guardar todos los datos en bases de datos, de tal forma que podamos gestionar la información de forma adecuada, con todas las garantías que nos da este tipo de software. Para la finalidad de lo que así se explica consideraremos que tendremos una tabla por cada tipo de objeto, cuyos campos serán las características (tanto comunes como particulares) de ese tipo de objeto. De esta forma la información asociada a cada objeto se guardará en la tabla del tipo al que pertenezca, por ejemplo los libros se guardarán en la tabla 'Libros'.

Evidentemente todo esto es una simplificación pero nos sirve para nuestro pequeño.

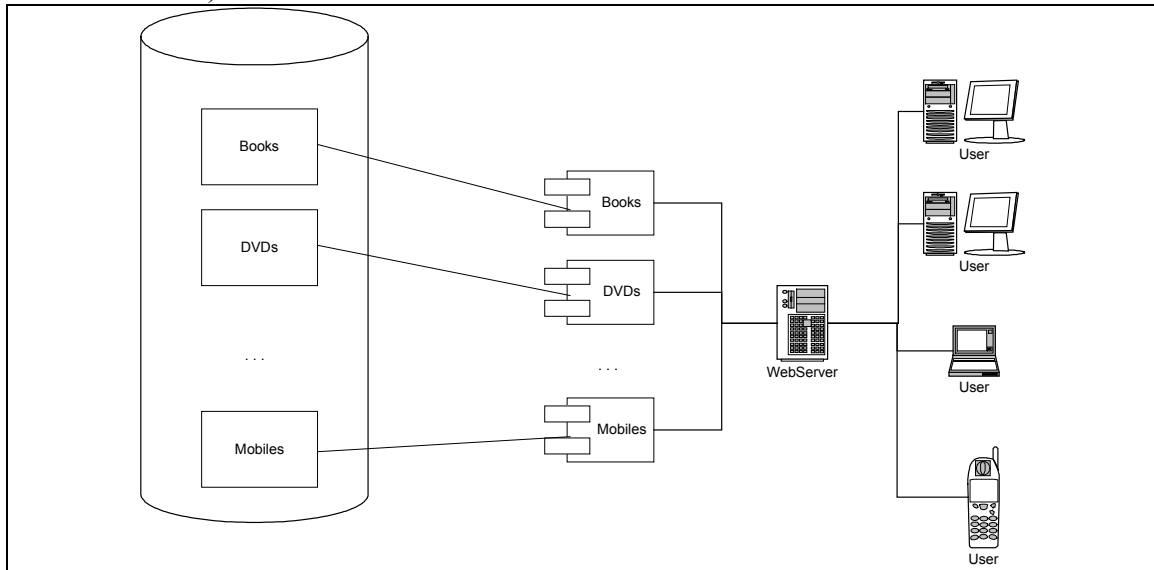
Nuestra tarea consiste básicamente en gestionar la publicación web de toda esta información (olvidemos el carrito de la compra). Publicar esta información consiste básicamente en permitir al usuario, a través de un navegador web, el poder:

- Visualizar listados de productos de un determinado tipo.
- Filtrar estos listados.
- Visualizar los detalles de un producto específico.

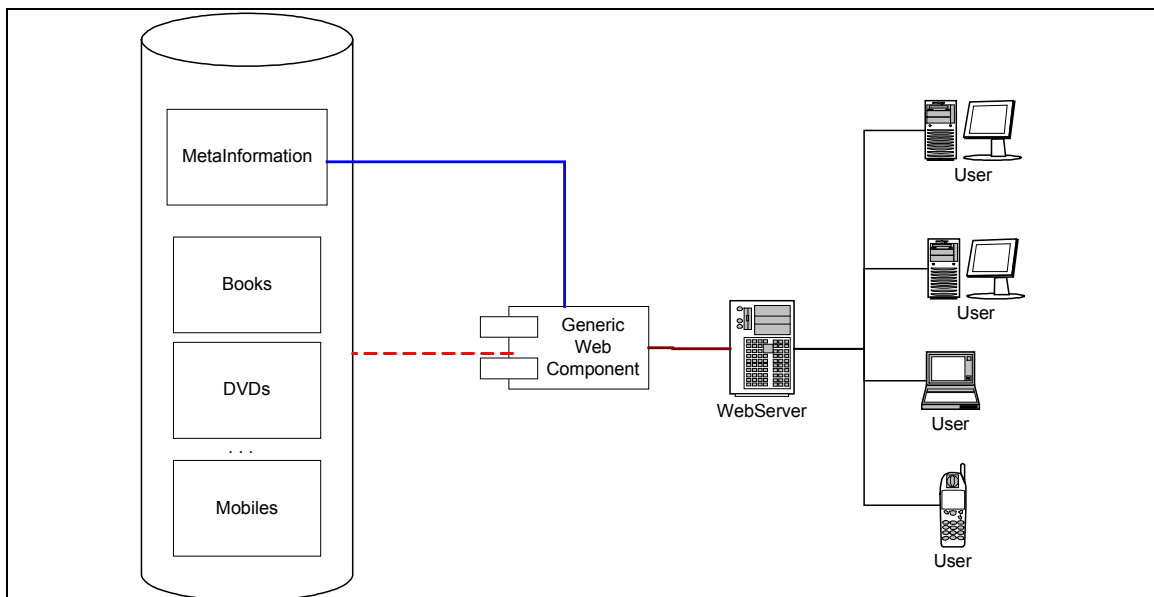
Tenemos básicamente dos posibilidades técnicas para llevar a buen puerto esta tarea:

- Construir o generar componentes web para cada una de estas categorías. De esta forma tendríamos un 'componente' (conjunto de páginas JSPs o ASPs) distinto para cada tipo de objetos. De esta forma encapsularíamos en cada componente el conocimiento sobre las características de cada tipología. Tendríamos por ejemplo un Jsp/Asp para listar DVDs (ListDvd.jsp), y otro para visualizar la

información asociada a un DVD (ShowDvd.jsp?id=83838). Esto implica tener mucho código que deberemos construir (costoso) o generar, y mantener (muy costoso).



- Tener un único componente web genérico de tal forma que leyendo las propiedades de cada tipo de objeto (datos que guardamos en la tabla MetaInformación), sea capaz de mostrar cualquier objeto sea cual sea su tipo. Podríamos verlo como tener un único Jsp/Asp para listados (List.jsp?type=DVD), y otro para visualizar registros (ShowElement.jsp?type=DVD&id=83838). A la información que describe la estructura de cada tipo de objeto se le llama metainformación. El coste de creación puede ser alto o no serlo, pero el de mantenimiento es muy bajo.



2.- Generación de Código Fuente (o desarrollo manual en su defecto).

Cada tipo de objeto tiene un conjunto de características comunes con otros tipos así como otras características diferentes al resto. Esto implica que en la base de datos tendremos tablas asociadas a cada tipo de objeto, y que por lo tanto los componentes web que listan y visualizan cada tipo (en definitiva páginas JSPs, ASPs, Servlets, etc) deben tener una estructura muy similar ya que su función es prácticamente la misma, pero a su vez esta estructura o código debe ser algo diferente debido a las características particulares de cada tabla (tipo de objeto) a la que sirve de interfaz web.

De hecho el código de cada componente asociado a un tipo es totalmente dependiente de la estructura de la tabla a la que sirve de interfaz, con lo que una vez definida cada tabla podríamos optar por generar de forma automatizada todo o parte del código que la lista, visualiza, etc, la otra posibilidad es utilizar la conocida técnica de copiar-pegar-modificar cuyos resultados suelen ser similares a la de la generación de código pero que es mucho más costosa y propensa a errores.

Existen multitud de herramientas que realizan funciones de generación de código, como por ejemplo [CodeCharge](#), [Xcripiter](#), y montones de wizards de diferentes entornos de desarrollo.

Ya que con la generación automática de código fuente obtenemos los mismos resultados, pero a un coste bajísimo, podríamos pensar que son la solución al problema que hemos planteado al inicio del artículo (la construcción de 'AnotherAmazon.com'). Sin embargo la generación de código tiene numerosas pegadas (no digamos ya la técnica de copiar-pegar-modificar) y por lo tanto no es la técnica apropiada para todos los proyectos.

Centrémonos por tanto en la generación de código fuente.

La generación de código tiene un escenario típico que es el de las interfaces entre diferentes capas, componentes, etc, lugares en los que una pequeña cantidad de información puede ser usada para generar todo el código repetitivo que se requiere (ej: Wsdl → Soap, mapeo OO → RDBMS, tablas → formularios) (en nuestro caso de tablas a listados Html y visualización Html de sus registros).

Existen básicamente 3 tipos de generación de código [ACG]:

- **Templating.** Se genera un armazón (o esbozo) de código fuente no funcional para ser editado, con el que se evita tener que escribir la parte más repetitiva del código (generalmente poco compleja). Suele ser una opción recomendable. Ejemplo: Si usamos Oracle JDeveloper para construir un nuevo Servlet, éste nos genera automáticamente una clase 'Servlet' con los imports básicos y los métodos que tiene esta clase.
- **Parcial.** Se genera código fuente que implementa parcialmente la funcionalidad requerida, pero que el programador usará como base para modificar, integrar y/o adaptar a sus necesidades. No suele ser recomendable. Por ejemplo: generamos una aplicación para el mantenimiento de tablas de bbdd (como aplicaciones web realizadas con CodeCharge, o componentes BC4J del JDeveloper), en el momento en que queramos modificarla, integrarla con otros desarrollos o adaptarla a necesidades ligeramente diferentes, tenemos que bucear en enormes cantidades de código que no entendemos y que debemos modificar extensivamente.
- **Total.** Se genera código fuente funcionalmente completo pero que no va a ser modificado por el programador, sino que si es necesario se vuelve a regenerar. Por lo general tampoco suele ser un código excesivamente complejo.

Recomendable. Ejemplo: generación de un Stub para un Webservice en JDeveloper.

Centrémonos en la generación parcial (es decir generación automática + modificaciones y desarrollos en el mismo). Suele ser problemática, ya que:

- El código generado es lo suficientemente complicado como para ser difícil de modificar con respecto a la funcionalidad implementada.
- Su regeneración suele ser costosa ya que hay si regeneramos el código hay que volver a modificarlo para volver a adaptarla al problema.
- Todo este código acaba siendo código a mantener, y no podemos olvidar que el mantenimiento es el mayor coste de cualquier proyecto.
- Hace que el código sea dependiente de una herramienta adicional de desarrollo. Esto es muy restrictivo, ya que complica el mantenimiento (en muchísimas ocasiones no lo hace la misma empresa que lo desarrolló, o surgen nuevos entornos que hacen inviable el uso de esas herramientas de generación (por ejemplo CodeCharge al principio generaba JSPs, ahora Servlets!!)), es una dependencia más y además fuerte.
- La generación de código funcional (parcial o semitotal) suele darte casi la funcionalidad que necesitas, pero no exactamente la necesitada. Modificarla es difícil.

Una de las grandes metas de la ingeniería del software es crear software reutilizable, lo cual suele traducirse en código robusto, poco propenso a errores.

Entonces, ¿por qué no parametrizar todo ese código repetitivo pero diferente unificándolo en uno solo (es decir refactorizarlo)? Todo depende de las dimensiones de la aplicación, ya que esa refactorización se consigue generalmente mediante arquitecturas reflexivas. No olvidemos que desarrollar una arquitectura tiene su coste.

Sólo hay dos aspectos positivos de la generación parcial de código:

- Por lo general tiene un mejor rendimiento respecto a las a.r.
- Es más rápido de desarrollar que construir una a.r. (a no ser que ya la tengas implementada).

La generación de componentes webs (JSPs, ASPs, etc) asociados a tipos de objetos entraría en la categoría de generación parcial de código. Es evidente que podemos generar código que liste y visualice registros de tablas, pero este código generalmente forma parte de una aplicación que debe de integrarse con desarrollos específicos como código de seguridad, uso de tecnologías específicas de acceso a base de datos (por ejemplo un pool de conexiones), elementos de navegación, normas de estilo y visualización, uso de logs, mecanismos de personalización, servicios de impresión, etc, etc. Cuando nos enfrentamos a la construcción de aplicaciones medianas/grandes la generación de código suele ser muy problemática.

3.- Qué es una arquitectura reflexiva.

Tal y como aparece en [POSA] las arquitecturas reflexivas se han usado en multitud de sistemas y son especialmente apropiadas cuando necesitamos aplicaciones muy flexibles y adaptativas.

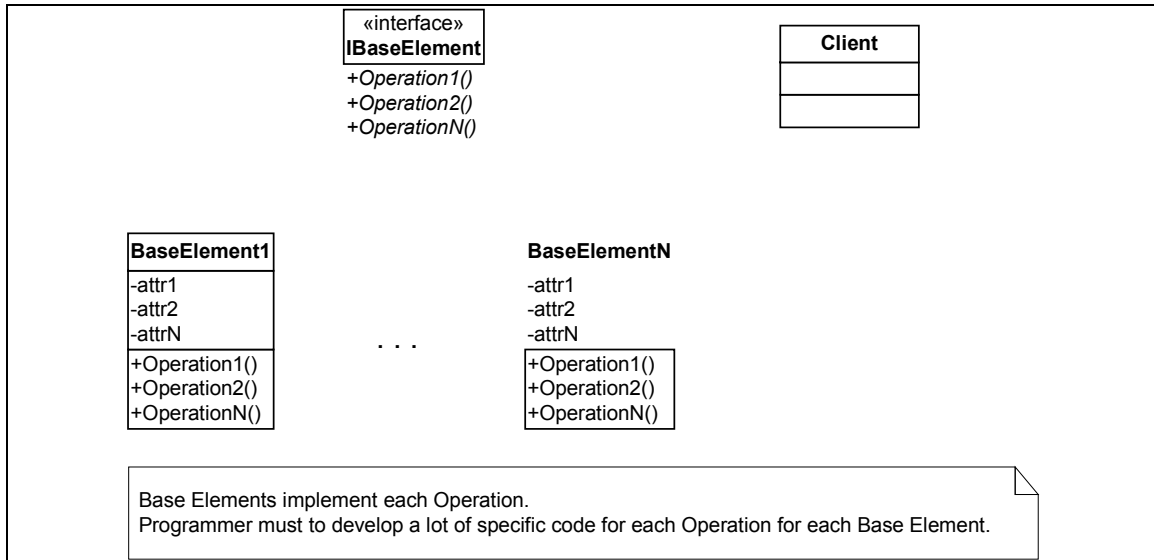
Una arquitectura reflexiva se caracteriza por tener una serie de elementos que contienen información estructural de parte del sistema, es decir que lo describe. El sistema usa esta metainformación como elemento básico que le permite interactuar con esos elementos descritos (generalmente llamados base).

Una característica fundamental es que por lo general todos los elementos base que son descritos comparten una serie de características de tal forma que pueden ser sometidos a ciertas transformaciones (operaciones) de forma uniforme, y que sin embargo también

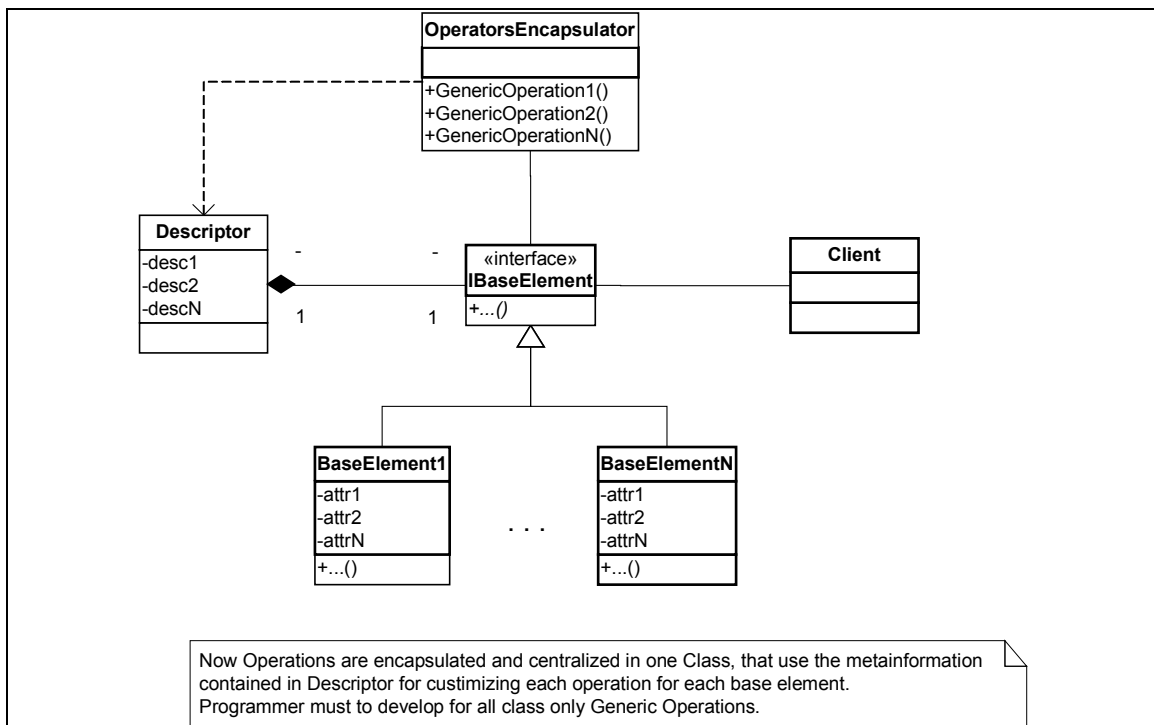
Arquitecturas Reflexivas y Generación de Código

contienen elementos diferenciadores que hace que estas operaciones sean bastantes dependientes de cada elemento base.

Estas características diferenciadas hacen que por lo general se desarrolle código específico por cada uno de estos elementos base, sin embargo existen ocasiones en las que nos es mucho más rentable desarrollar una arquitectura reflexiva que trate a todos estos elementos de forma uniforme.



Toda esta arquitectura puede verse también como una típica estrategia de centralización o aplicación del metapatrón encapsulador-centralizador.



Beneficios:

- Cambiar un sistema software (añadir elementos, campos, etc) es fácil.
- Soporta numerosos tipos de cambios.

Perjuicios:

- Menor eficiencia.
- Se incrementa el número de componentes.

Las arquitecturas reflexivas no siempre son la mejor opción, sobre todo si no hay muchos elementos base de los que generar código.

Respecto al rendimiento, ciertamente son arquitecturas que añaden una capa adicional de indirección y que por lo tanto dependiendo de la aplicación específica pueden ser lentas (aunque no tienen por qué serlo por norma general). Existen además medios para aumentar espectacularmente el rendimiento de esta arquitectura, como por ejemplo el uso de cachés.

4.- Cuándo desarrollar una arquitectura reflexiva.

Desarrollar una arquitectura tiene costes y no es apropiada en todas las circunstancias, sino que dependen de numerosos factores.

Por lo general implementar una arquitectura reflexiva requiere desarrollar elementos de código bastante complejo, necesita de ciertas clases adicionales, de ese nivel adicional de indirección que es precisamente el que nos proporciona la potencia y flexibilidad necesario.

Por esto si la aplicación es pequeña no es aconsejable implementar una arquitectura reflexiva, ya que ésta elevaría la complejidad de la aplicación, su coste y los conocimientos necesarios para su mantenimiento sin aportar mucho a cambio.

Por tanto, ¿cuándo implementarla?:

- Precisamente cuando necesitamos un sistema altamente flexible en ciertos aspectos (ya sea por su dinámica en explotación o en su desarrollo, desconocimiento de requisitos o que sean muy difusos).
- Cuando tengamos una gran cantidad de elementos base (como por ejemplo tablas) en los que la otra opción sería la generación de enormes cantidades de bloques de código similar pero altamente dependiente del objeto al que sirven de interfaz funcional (por ejemplo tablas de bases de datos).
- Cuando nuestra actividad técnica consista en realizar aplicaciones en las que por lo general se utilicen técnicas de copiar-pegar-modificar o generación parcial de código. En este caso el desarrollo de una arquitectura reflexiva que reutilicemos en todos nuestros desarrollos queda plenamente justificada y amortizada.

Imaginemos un portal como el que hemos descrito al inicio de este artículo (AnotherAmazon.com) que gestione numerosas tablas distintas de objetos para su venta, es decir aquí existirían una gran cantidad de elementos base. Este es el escenario ideal para implementar una arquitectura reflexiva que además nos permita modificar de una forma centralizada y sencilla tanto la información que contienen estos elementos, añadir nuevos elementos de información como por ejemplo 'Reproductores DVDs', etc.

Referencias:

- [ACG] "Automated Code Generation":
<http://c2.com/cgi/wiki?AutomatedCodeGeneration>
- [CGIDS] "Code Generation is a Design Smell":
<http://c2.com/cgi/wiki?CodeGenerationIsaDesignSmell>
- [OAOO] "Only and Only Once":
<http://c2.com/cgi/wiki?OnceAndOnlyOnce>
- [RCG] "Reflection versus Code Generation":
<http://www.javaworld.com/javaworld/jw-11-2001/jw-1102-codegen.html>
- [POSA] Pattern-Oriented Software Architecture: A System of Patterns (POSA) Buschmann, Meunier, Rohnert, Sommerlad, Stal; Wiley and Sons, 1996.
- [WIR] "What is Refactoring"
<http://c2.com/cgi/wiki?WhatIsRefactoring>