**Patrones de Diseño en Java**

Como analistas y programadores vamos desarrollando a diario nuestras habilidades para resolver problemas usuales que se presentan en el ***desarrollo del software***. Por cada problema que se nos presenta pensamos distintas formas de resolverlo, incluyendo soluciones exitosas que ya hemos usado anteriormente en problemas similares. Es así que a mayor experiencia que tengamos, nuestro abanico de posibilidades para resolver un problema crece, pero al final siempre habrá una sola solución que mejor se adapte a nuestra aplicación. Si documentamos esta solución, podemos reutilizarla y compartir esa información que hemos aprendido para resolver de la mejor manera un problema específico.

Los **patrones del diseño** tratan los problemas del diseño que se repiten y que se presentan en situaciones particulares del diseño, con el fin de proponer soluciones a ellas. Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes. Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones son llamadas estrategias.

En este artículo, veremos algunos patrones de diseño para las aplicaciones basadas en **J2EE**.

Breve historia de los patrones de diseño

Un *patrón de diseño* es una abstracción de una solución en un nivel alto. Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que abarcan las distintas etapas del desarrollo; desde el **análisis hasta el diseño** y desde la **arquitectura** hasta la **implementación**.

Muchos diseñadores y arquitectos de *software* han definido el término de ***patrón de diseño*** de varias formas que corresponden al ámbito a la cual se aplican los patrones. Luego, se dividió los patrones en diferentes categorías de acuerdo a su uso.

Los diseñadores de software extendieron la idea de **patrones de diseño** al proceso de desarrollo de software. Debido a las características que proporcionaron los *lenguajes orientados a objetos* (como herencia, abstracción y encapsulamiento) les permitieron relacionar entidades de los lenguajes de programación a entidades del mundo real fácilmente, los diseñadores empezaron a aplicar esas características para crear soluciones comunes y reutilizables para problemas frecuentes que exhibían patrones similares.

Fue por los años 1994, que apareció el libro "*Design Patterns: Elements of Reusable Object Oriented Sofware*" escrito por los ahora famosos **Gang of Four** (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Ellos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos. Desde luego que ellos no son los inventores ni los únicos involucrados, pero ese fue luego de la publicación de ese libro que empezó a difundirse con más fuerza la idea de patrones de diseño.

El grupo de **GoF** clasificaron los patrones en 3 grandes categorías basadas en su **PROPÓSITO**: creacionales, estructurales y de comportamiento.

* **Creacionales**: Patrones creacionales tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
* **Estructurales**: Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.
* **Comportamiento**: Los patrones de comportamiento nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

En el segundo nivel, ellos clasificaron los patrones en 2 ámbitos: Clases y objetos. Es así que, tenemos 6 tipos de patrones:

|  |  |
| --- | --- |
| **Creacionales** | * **Creacional de la Clase** Los patrones creacionales de Clases usan la herencia como un mecanismo para lograr la instanciación de la Clase. Por ejemplo el método Factoría. * **Creacional del objeto** Los patrones creacionales de objetos son más escalables y dinámicos comparados de los patrones creacionales de Clases. Por ejemplo la Factoría abstracta y el patrón Singleton. |
| **Estructurales** | * **Estructural de la Clase** Los patrones estructurales de Clases usan la herencia para proporcionar interfaces más útiles combinando la funcionalidad de múltiples Clases. Por ejemplo el patrón Adaptador (Clase). * **Estructural de Objetos** Los patrones estructurales de objetos crean objetos complejos agregando objetos individuales para construir grandes estructuras. La composición de l patrón estructural del objeto puede ser cambiado en tiempo de ejecución, el cual nos da flexibilidad adicional sobre los patrones estructurales de Clases. Por ejemplo el Adaptador (Objeto), Facade, Bridge, Composite. |
| **Comportamiento** | * **Comportamiento de Clase** Los patrones de comportamiento de Clases usan la herencia para distribuir el comportamiento entre Clases. Por ejemplo Interpreter. * **Comportamiento de Objeto** Los patrones de comportamiento de objetos nos permite analizar los patrones de comunicación entre objetos interconectados, como objetos incluidos en un objeto complejo. Ejemplo Iterator, Observer, Visitor. |

Patrones J2EE

Con la aparición del **J2EE**, todo un nuevo catálogo de patrones de diseño apareció. Desde que **J2EE** es una arquitectura por si misma que involucra otras arquitecturas, incluyendo servlets, *JavaServer Pages*, *Enterprise JavaBeans*, y más, merece su propio conjunto de patrones específicos para diferentes aplicaciones empresariales.

De acuerdo al libro "*J2EE PATTERNS Best Practices and Design Strategies*", existen 5 capas en la arquitectura J2EE:

* Cliente
* Presentación
* Negocios
* Integración
* Recurso

El libro explica 15 patrones J2EE que están divididos en 3 de las capas: presentación, negocios e integración.

Catálogo de patrones J2EE

Capa de Presentación

|  |  |
| --- | --- |
| **Decorating Filter / Intercepting Filter** | Un objeto que está entre el cliente y los componentes Web. Este procesa las peticiones y las respuestas. |
| **Front Controller/ Front Component** | Un objeto que acepta todos los requerimientos de un cliente y los direcciona a manejadores apropiados. El patrón Front Controller podría dividir la funcionalidad en 2 diferentes objetos: el *Front Controller* y el *Dispatcher*. En ese caso, El Front Controller acepta todos los requerimientos de un cliente y realiza la autenticación, y el Dispatcher direcciona los requerimientos a manejadores apropiada. |
| **View Helper** | Un objeto helper que encapsula la lógica de acceso a datos en beneficio de los componentes de la presentación. Por ejemplo, los *JavaBeans* pueden ser usados como patrón View Helper para las páginas JSP. |
| **Composite view** | Un objeto vista que está compuesto de otros objetos vista. Por ejemplo, una página *JSP* que incluye otras páginas JSP y *HTML* usando la directiva include o el action include es un patrón Composite View. |
| **Service To Worker** | Es como el patrón de diseño MVC con el Controlador actuando como Front Controller pero con una cosa importante: aquí el Dispatcher (el cual es parte del Front Controller) usa View Helpers a gran escala y ayuda en el manejo de la vista. |
| **Dispatcher View** | Es como el patrón de diseño MVC con el controlador actuando como Front Controller pero con un asunto importante: aquí el Dispatcher (el cual es parte del Front Controller) no usa View Helpers y realiza muy poco trabajo en el manejo de la vista. El manejo de la vista es manejado por los mismos componentes de la Vista. |

Capa de Negocios

|  |  |
| --- | --- |
| **Business Delegate** | Un objeto que reside en la capa de presentación y en beneficio de los otros componentes de la capa de presentación llama a métodos remotos en los objetos de la capa de negocios. |
| **Value Object/ Data Transfer Object/ Replicate Object** | Un objeto serializable para la transferencia de datos sobre lar red. |
| **Session Façade/ Session Entity Façade/ Distributed Façade** | El uso de un bean de sesion como una fachada (facade) para encapsular la complejidad de las interacciones entre los objetos de negocio y participantes en un flujo de trabajo. El Session Façade maneja los objetos de negocio y proporciona un servicio de acceso uniforme a los clientes. |
| **Aggregate Entity** | Un bean entidad que es construido o es agregado a otros beans de entidad. |
| **Value Object Assembler** | Un objeto que reside en la capa de negocios y crea Value Objets cuando es requerido. |
| **Value List Handler/ Page-by-Page Iterator/ Paged List** | Es un objeto que maneja la ejecución de consultas SQL, caché y procesamiento del resultado. Usualmente implementado como beans de sesión. |
| **Service Locator** | Consiste en utilizar un objeto Service Locutor para abstraer toda la utilización JNDI y para ocultar las complejidades de la creación del contexto inicial, de búsqueda de objetos home EJB y recreación de objetos EJB. Varios clientes pueden reutilizar el objeto Service Locutor para reducir la complejidad del código, proporcionando un punto de control. |

Capa de Integración

|  |  |
| --- | --- |
| **Data Access Object Service Activator** | Consiste en utilizar un objeto de acceso a datos para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos. |
| **Service Activator** | Se utiliza para recibir peticiones y mensajes asíncronos de los clientes. Cuando se recibe un mensaje, el Service Activator localiza e invoca a los métodos de los componentes de negocio necesarios para cumplir la petición de forma asíncrona. |