UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

CREACIÓN DE FRAMEWORKS CON PATRONES DE DISEÑO PARA EL DESARROLLO DE APLICACCIONES EMPRESARIALES

TESIS PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A :

CUAUHTEMOC HOHMAN SANDOVAL

DIRECTOR DE TESIS
ING. CARLOS ALBERTO ROMÁN ZAMITIZ

CIUDAD UNIVERSITARIA
MÉXICO, D.F. 2014

CREACIÓN DE FRAMEWORKS CON PATRONES DE DISEÑO PARA EL DESARROLLO DE APLICACIONES EMPRESARIALES

CONTENIDO

I. INTRODUCCIÓN

A finales de la década de los 60, se introdujo al mundo de la informática el llamado hardware de tercera generación, que hacía uso de los revolucionarios circuitos integrados (microchips). Estos permitieron no sólo construir máquinas con mucho más poder de procesamiento y capacidad de almacenamiento, sino también construir otras mucho más pequeñas y baratas, las minicomputadoras.

Las minicomputadoras abrieron la entrada al mercado a nuevos competidores en la industria del hardware, y el incremento en la oferta, permitió que más personas tuvieran acceso a las computadoras, lo que a su vez incrementó la demanda y ello impulsó aún más el desarrollo del hardware.

Sin embargo, fue durante este ciclo de bonanza, cuando se acuñó el término “crisis del software”, que se refiere a que por primera vez en la industria de la informática, el hardware no era la principal limitación. La causa principal por la que los proyectos de informática no resultaban ser exitosos era el desarrollo del software.

Con un hardware más poderoso se podían atacar problemas más complejos y variados que antes, pero la dificultad de desarrollar programas de computadora bien escritos, entendibles y verificables los limitó. Como consecuencia los proyectos de informática no terminaban en plazo, no se ajustaban al presupuesto inicial, el software generado era de baja calidad, muy ineficiente, no cumplía las especificaciones y el código era difícil de mantener, lo que complicaba la gestión y evolución del proyecto, y en muchas ocasiones el software nunca se entregó.

La cuarta generación de computadoras, a mediados de los años 70, acentuó aún más los problemas en el software. Se desarrolló el microprocesador y así se construyeron las primeras microcomputadoras, que con el tiempo generalizaron el uso de las computadoras entre la población, ya no limitándo su uso, a centros de investigación y grandes empresas.

Esta crisis estimuló el desarrollo de la disciplina conocida como ‘Ingeniería de Software’, que a lo largo de las últimas décadas ha identificado muchos de los problemas que se presentan al crear software, y encontrar soluciones ha sido una prioridad para muchos investigadores y empresas del ramo.

Mediante la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, se crearon muchas herramientas, métodos formales, procesos, mejores prácticas, códigos de profesionalismo y disciplina, que en su momento se anunciaban como “la solución” al problema.

Lo cierto es que estas herramientas y metodologías han mejorado parcialmente el desarrollo del software pero sin llegar a soluciones completas. La ingeniería del software es un proceso complejo presenta diversas soluciones. Cada proyecto es diferente, pero eligiendo las herramientas más apropiadas de acuerdo al contexto, se puede llevar a buen término un proyecto de software.

En la actualidad, el creciente uso de las tecnologías de la información ha aumentado la necesidad, variedad y complejidad del software. Los sistemas deben tener interfaces gráficas más elaboradas, interactuar constantemente con otros sistemas, tener tiempos

de respuesta cortos, usarse en una variedad de dispositivos, cumplir con las exigencias de grupos de usuarios más grandes y diversos, etcétera.

Un sistema grande puede contener tantas partes móviles y cambiantes que puede ser difícil para una persona llevar el registro de sus interrelaciones. Incluso con un sistema reducido, sería imposible para un desarrollador de software cumplir con estas exigencias si debe inventar la rueda cada que empieza un nuevo desarrollo. En vez de empezar de cero, debe apoyarse de la experiencia e infraestructura que otros desarrolladores ya hayan implementado.

Por otro lado el cambio es inherente al software computacional, sin importar la etapa en que éste se encuentre, el sistema cambiará y el deseo de cambiarlo persistirá durante todo su ciclo de vida, ya sea por nuevas condiciones en el negocio, nuevas necesidades del cliente, una reorganización o crecimiento del negocio, restricciones presupuestales, de recursos u otra índole.

Se deben desarrollar sistemas preparados para el cambio, en vez de intentar oponerse a él, de tal forma que reduzcan el esfuerzo para implementar dichos cambios y minimicen los impactos no deseados sobre el resto del sistema.

En todo este tiempo, el tema ha cobrado una gran relevancia y se le han destinado incontables recursos para resolver los problemas en la planeación, el desarrollo y la implementación de un proyecto de software, de tal suerte que hoy en día existen numerosas herramientas con las que cuenta un ingeniero de software, entre las que podemos mencionar nuevos lenguajes de programación a alto nivel, técnicas de modelado, herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Computadora), estándares de codificación, metodologías de desarrollo, frameworks, sistemas de control de versiones, entre otros.

El ingeniero de software debe elegir cuidadosamente las herramientas más adecuadas al proyecto, dadas las características del mismo y el ambiente en el que se encuentra. En esta tesis nos enfocamos en el uso y creación de una de esas herramientas, los Frameworks, ya que ayudan a facilitar el diseño de la arquitectura y a construir software de forma más rápida y eficiente, al heredar de ellos la experiencia, las estructuras y piezas prefabricadas que el framework proporcione.

El principal objetivo de esta tésis es desarrollar la base teórica sobre la que, por su dificultad, el diseño y la creación de un framework debería estar fundada.

En segundo término, a modo de caso de estudio, se plantea la creación de un framework que ayude a programar ciertos aspectos comunes en una aplicación empresarial.

II. FRAMEWORKS

DEFINICIÓN DE FRAMEWORK

La palabra inglesa "**framework**" (marco de trabajo) define, en términos generales, un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

En el desarrollo de software, un ***framework*** o ***infraestructura digital***, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software.

Un framework es una aplicación reutilizable y semi-completa que puede ser especializada para producir aplicaciones individualizadas, ya que es una abstracción de software que proporciona una funcionalidad genérica que debe ser cambiada de forma selectiva por el código adicional escrito por el usuario, lo que genera el **software de aplicación** específica.

Los frameworks pueden incluir programas de apoyo, compiladores, bibliotecas de código, juegos de herramientas e interfaces de programación de aplicaciones (API, por sus siglas en inglés) que reúnen a todos los diferentes componentes para permitir el desarrollo de un proyecto o solución.

Un framework debe resolver alguna situación repetitiva que quite mucho tiempo de programación o que complique la implementación, de forma tal que el desarrollador pueda concentrarse en implementar la lógica de negocio particular de la aplicación, permitiendo que el framework se encargue de la plomería y las tareas comunes de toda aplicación.

REUTILIZACIÓN DE CÓDIGO

La reutilización de código se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un programa informático existente se pueda emplear en la construcción de otro programa. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce la redundancia.

La manera más sencilla de reutilizar código es copiar el código fuente, total o parcialmente, desde el programa antiguo al programa en desarrollo. Pero resulta que aunque esta técnica es sencilla de llevar a cabo, a largo plazo cuesta mucho trabajo mantener múltiples copias del mismo código.

Para eliminar la duplicidad de código existe otra técnica conocida como bibliotecas de software, en las que se agrupa el código reusable en un único lugar accesible desde los diferentes programas. Se agrupan varias operaciones comunes a cierto dominio para facilitar el desarrollo de programas nuevos.

Existen, por ejemplo, bibliotecas para convertir información entre diferentes formatos conocidos, acceder a dispositivos de almacenamiento externos, proporcionar una interfaz

con otros programas, manipular información de manera conocida (como números, fechas, o cadenas de texto), etcétera.

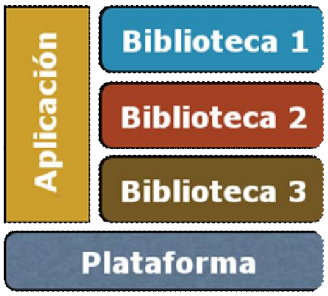

FIGURA II.1 - REPRESENTACIÓN DE UNA APLICACIÓN HACIENDO USO DE UNA LIBRERÍA

Para que el código existente se pueda reutilizar, debe definir alguna forma de comunicación o interfaz. Esto se puede dar por llamadas a una subrutina, a un objeto, o a una clase.

Los frameworks orientados a objetos son otra técnica que permite reutilizar diseños e implementaciones de software que han probado reiteradamente su efectividad, de forma que reducen el costo y mejoran la calidad de la aplicación.

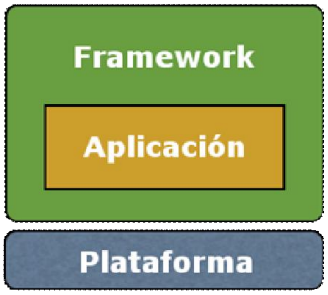

FIGURA II.2 - REPRESENTACIÓN DE UNA APLICACIÓN HACIENDO USO DE UN FRAMEWORK

En contraste con las técnicas iniciales, basadas en bibliotecas que contienen funciones o subrutinas, los frameworks son más intrusivos, ya que pueden definir directamente la arquitectura de la aplicación y el orden en que se ejecutan las subrutinas, mientras que con las bibliotecas, es el programador, el que mantiene todo el control y determina el momento de hacer uso de una subrutina.

Características distintivas que separan a los frameworks de las bibliotecas tradicionales:

1. Inversión de control - En un framework, a diferencia de en las bibliotecas o aplicaciones de usuario normales, el control de flujo del programa general no está dictada por la persona que llama, sino por el framework.

2. Comportamiento por defecto - Un framework tiene un comportamiento predeterminado. Este comportamiento por defecto debe ser una conducta útil y no una serie de operaciones no soportadas.

3. Extensibilidad - El programador usuario del framework debe poder extender la funcionalidad del mismo de forma tal que satisfaga la funcionalidad específica de

la aplicación que construye. Por lo general esto lo logra implementando interfaces y sobrescribiendo métodos definidos en el framework para tal fin.

4. Código no modificable - En general, el código del framework no debe ser modificado por el usuario, excepto por los usos permitidos por la extensibilidad.

Hoy en día los frameworks juegan un papel cada vez más importante en el desarrollo de software, ya que los vertiginosos avances en la tecnología generan cambios importantes en el mercado y en la forma de hacer negocios.

El cambio es la única constante en el desarrollo de software, "sin importar donde se trabaje, el tipo de sistema que se construya, el lenguaje de programación utilizado o lo bien planeada que este una aplicación, con el tiempo una aplicación debe crecer y cambiar o morir."

La intención de los frameworks es reducir el tiempo requerido para crear una nueva aplicación, al tiempo que aumenta la calidad de la misma para reducir el número de defectos y facilitar el mantenimiento, que incluye corregir errores y desarrollar nueva funcionalidad. Todo ello al proporcionar un marco de desarrollo que determine una arquitectura y una variedad de reglas, que son resultado de su uso extensivo y exitoso en una variedad de sistemas.

DISEÑO DE FRAMEWORKS

El diseño de un framework no es una tarea sencilla, hay muchos factores a considerar en la etapa de diseño, puesto que el diseño puede significar el éxito o el fracaso de su uso en un proyecto de desarrollo de una aplicación.

- Establecer los requerimientos
- Determinar el alcance del framework
- Mantener un diseño abstracto
- Diseñar la jerarquía de clases
- Crear un framework flexible

Típicamente un framework es implementado con un lenguaje orientado a objetos como lo es C++, Smalltalk o Java, ya que aprovechan las tres características distintivas de estos lenguajes:

- Abstracción de datos: Como en un tipo de dato abstracto, una clase abstracta representa una interfaz detrás de la cual la implementación puede cambiar sin afectar a los componentes que hacen uso de ella.
- Polimorfismo: Habilidad de una variable o parámetro de poder recibir valores de distintos tipos. Ello permite al desarrollador, entre otras cosas, definir métodos que pueden trabajar con una variedad de objetos diferentes o cambiar los colaboradores (objetos que interactúan) de un componente en tiempo de ejecución.
- Herencia: Facilita la creación de nuevos componentes basados en otros prexistentes.

Cada objeto en el framework es descrito por una clase abstracta o interfaz. Una clase abstracta es una clase que no puede tener instancias, así que sólo puede ser usada como una plantilla para crear subclases. Usualmente una clase abstracta tiene al menos una operación no implementada que se deja a sus subclases para extender e implementar los componentes.

En los lenguajes orientados a objetos más recientes, como Java, COM o CORBA, se separa las interfaces de las clases. En estos sistemas, un framework puede ser descrito únicamente en términos de sus interfaces. Sin embargo, estos sistemas sólo pueden especificar los aspectos estáticos de una interfaz, pero frecuentemente un framework incluye también el modelo colaborativo y los patrones de interacción. Consecuentemente, es común para un framework java el tener tanto interfaces como clases abstractas definidas para un componente.

Características de un framework bien diseñado:

- Simple
- Claro
- Limitado
- Flexible

Simple: La estructura externa del framework debe ser fácil de entender. Un framework bien diseñado debe ser fácil de enseñar a un desarrollador nuevo, de forma que en poco tiempo pueda empezar a desarrollar cosas sencillas usando el framework aún sin entender a detalle el funcionamiento del framework. Con el tiempo y el uso, el desarrollador irá conociendo el API (los objetos, sus métodos y propiedades) para hacer un mejor y más eficiente uso del framework.

Esta simplicidad se logra al proveer a cada parte del framework una interfaz clara y consistente.

Claro: Todo el detalle del comportamiento debe ser encapsulado, ocultando al desarrollador los detalles y la complejidad que el framework usa para funcionar. El desarrollador debería poder usar el framework como una caja negra, en la que no conoce los detalles de lo que sucede dentro de ella, pero debe tener claro lo que debe de proveerle y lo que sucederá como consecuencia.

Si el desarrollador requiere de entender el código dentro del framework para poder usarlo, entonces en definitiva hay un problema de diseño en el framework.

La interfaz pública de las clases en el framework debe ser tan simple como sea posible, ya que es con éstas con las que el desarrollador estará trabajando, debiendo mantenerse tan complicado como sea estrictamente necesario para lograr la funcionalidad deseada.

Por ejemplo, debería evitarse la necesidad de que el desarrollador llame en cierto orden 3 ó 4 métodos para lograr una cierta funcionalidad, en vez de ello sería mejor tener un solo método que llame esos métodos en el orden apropiado.

Limitado: Un framework debe tener un alcance bien definido, puesto que el framework debe cumplir con eso y solamente eso. Durante el desarrollo del framework es tentador violar los límites de alcance en aras de facilitar el desarrollo al programador, sin embargo esto suele hacer más complejo el framework y puede en ocasiones, contrariamente a lo intencionado, ser un obstáculo al desarrollador (que deberá buscar soluciones alternativas para lograr sus objetivos).

Un framework no debe proveer funcionalidad para la aplicación, sólo debe proveer el esqueleto sobre el que esa funcionalidad es construida. La funcionalidad de la aplicación es responsabilidad del desarrollador que usa el framework y el framework no debe interponerse.

Si el desarrollador de un framework desea proveer componentes especializados para ser usados por el desarrollador de la aplicación, éstos deben ser proporcionados en librerías separadas, como subclases de las interfaces definidas en el framework. Esto permitirá al

desarrollador elegir entre usar o no estas clases especializadas y hace clara la distinción entre el framework y un toolkit para desarrollar.

Flexible – Extensible: Debe ser fácil para el desarrollador el expandir el framework al añadir nuevas clases o subclases. Un framework cuyo comportamiento no es fácilmente modificable restringe al desarrollador en vez de empoderarlo con una herramienta.

La única manera en que un framework sea flexible-extensible es que se le provea dicha habilidad desde el diseño del mismo, difícilmente esto se puede añadir después.

Por su naturaleza un framework debe ser muy flexible, adaptable a una variedad de sistemas, y para lograrlo suele entonces hacer uso de una variedad de patrones de diseño.

Una técnica común para proveer de flexibilidad es el uso de “ganchos” (hooks), que básicamente son métodos definidos en el framework, pero sobrescritos en subclases escritas por el desarrollador de la aplicación. El framework controla las llamadas a estos métodos, de forma que sean llamados en el momento apropiado. Por ejemplo, si un formulario tuviera un método guardar(), el framework podría definir convenientemente otro par de métodos antesDeGuardar() y despuesDeGuardar(), tales que se ejecuten respectivamente antes y después de ejecutar el método guardar(), de tal forma que el desarrollador de la aplicación pueda sobrescribir dichos métodos para realizar validaciones antes de guardar, en caso de error interrumpir el proceso de guardado y mostrar un mensaje al usuario con el resultado después del guardado.

Este tipo de ganchos pueden, y deben, ser provistos donde sea que hagan sentido y permitan al desarrollador introducir la lógica de aplicación requerida.

VENTAJAS

Utilizar frameworks en el desarrollo de un sistema proporciona una serie de ventajas:

Reutilización de código: Los frameworks son desarrollados específicamente para ser usados en una variedad de proyectos, por lo que una ventaja principal es que el programador que lo usa, está reutilizando el código que otras personas se tomaron la molestia en diseñar, codificar y probar.

Código probado: El framework no sólo es probado por los desarrolladores que lo codificaron, sino que es probado continuamente por toda la comunidad de programadores que hace uso de él en situaciones muy variadas. Esto da confianza al programador para utilizarlo y saber que existe información generada por otros usuarios que alertan sobre posibles fallos en el framework y hasta la forma de evadirlos.

Facilita la creación de una comunidad de usuarios alrededor del framework: La comunidad ayuda a difundir el framework (haciéndolo más conocido, lo que a su vez acrecienta y fortalece a la comunidad) mediante foros de discusión en internet, documentan, comparten experiencias y resolución de problemas e incluso desarrollan herramientas y extensiones que les hagan más fácil trabajar con él.

Implantación de mejores prácticas: En el diseño y construcción de un framework, los desarrolladores se basan de todo su conocimiento y experiencia adquirida a lo largo de muchos años. El resultado es que el programador que usa el framework utiliza, a veces sin saberlo, las mejores prácticas introducidas en él desde el diseño, tanto las que se usen internamente, como las expuestas en el API para uso y extensión.

Definición de arquitectura: Los frameworks definen (total o parcialmente) la estructura arquitectónica de la aplicación que se monte sobre ellos. Esto ayuda al arquitecto del sistema a tomar las decisiones más rápidamente, sobre la forma de organizar los módulos, componentes y demás recursos que en conjunto forman la aplicación.

Acelera el desarrollo de aplicaciones: Al sumar todos los puntos anteriores, el resultado final del uso de los frameworks apropiados en una aplicación es que debería recortarse el tiempo necesario para implementar el desarrollo.

Después de un análisis inicial del sistema a desarrollar, se puede tener información suficiente para elegir el o los frameworks que conviene utilizar, con lo que implícitamente se están tomando decisiones sobre la arquitectura y se tiene idea sobre las posibilidades y limitaciones que se tendrán. Una vez iniciado el desarrollo, los programadores se pueden concentrar en implementar la lógica del negocio, despreocupándose por todo el código de "plomería" que los frameworks les proporcionan.

DESVENTAJAS

No todo en el uso de frameworks es "miel sobre hojuelas", existen varios aspectos a tomar en cuenta al introducir un framework nuevo en un proyecto.

Curva de aprendizaje: Cada framework es muy particular, tiene un API diferente y formas distintas de hacer las cosas, por lo tanto existe un tiempo de aprendizaje por el que cada programador que quiera usar un framework tiene que pasar, similar al de aprender un lenguaje nuevo.

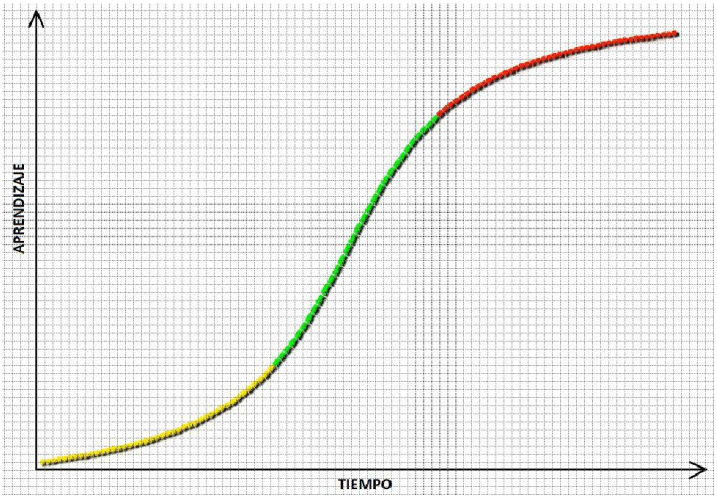

FIGURA II.3 - REPRESENTACIÓN DE UNA CURVA DE APRENDIZAJE

El inicio de este aprendizaje suele ser muy lento, durante esta etapa el programador debe dedicar todo su tiempo y esfuerzo en conocer la herramienta, por lo que no es de mucha utilidad para el proyecto.

En una segunda etapa, se acelera el aprendizaje y el programador puede realizar tareas sencillas en beneficio del proyecto. Esta experiencia acelera aún más su aprendizaje.

Finalmente, en una tercera etapa se domina el framework, el aprendizaje deja de ser pronunciado, pero es en este punto en el que se aprovechan las ventajas del framework aprendido y el programador puede aportar valor al proyecto.

Lo que esto implica es que un programador que no conoce el framework no puede comenzar inmediatamente a codificar, sino que debe dedicar tiempo a conocer la herramienta y eso puede retrasar un proyecto si no se le toma en cuenta.

Limitaciones y restricciones del framework: Los frameworks pueden llegar a ser muy intrusivos en el código de la aplicación ya que obligan a utilizar la arquitectura que define o a utilizar las clases e interfaces de las que debe heredar la aplicación. Esto crea un alto acoplamiento con las interfaces/clases del framework y hacen difícil quitar o cambiar el framework por otro con características similares.

Igualmente un framework puede restringir la forma de hacer una acción (normalmente por simplificar el proceso), pero pueden existir casos en los que este no es el comportamiento deseado y entonces hay que buscar la forma de romper con el paradigma de framework, lo que puede complicar el código en vez de simplificarlo.

Uso de un framework no apropiado: Es posible que se elija usar un framework no apropiado para el fin que se pretende alcanzar o bajo las circunstancias en las que se encuentra el proyecto por una variedad de razones.

El framework puede:

- No estar diseñado para el uso que se le pretende dar.
- No ser compatible con otros frameworks usados en el proyecto.
- No estar lo suficientemente probado y causar problemas en los que puede ser difícil detectar la causa (y más aún la solución) ya que el código del framework normalmente es ajeno al programador.
- Puede no tener un buen soporte de la comunidad. Una comunidad fuerte (en número de integrantes y participación) es importante para proporcionar soporte a los programadores, pues es común que los problemas con los que se enfrente una persona al usar un framework, se los encuentre otra después. Si existe falta de documentación, foros de discusión y otras herramientas, encontrar información útil será muy complicado.
- Puede no estar lo suficientemente maduro, por lo que puede haber cambios no compatibles o radicales entre una versión y la siguiente, lo que puede complicar introducir actualizaciones que corrigen errores en el framework o agregan mejoras en rendimiento o en su utilización.

EJEMPLOS DE FRAMEWORKS EN EL MERCADO

Hoy en día existen una gran cantidad de frameworks disponibles en la red. Este auge se explica porque realmente ayudan a desarrollar aplicaciones de una forma mejor y más rápida, evitando tener que inventar el hilo negro cada que se empieza un nuevo proyecto, y permitir que los programadores menos experimentados se beneficien de toda la experiencia que se introduce en el framework.

Sin embargo cada framework se enfoca en resolver problemáticas diferentes, o propone resolver las mismas problemáticas con otro paradigma y esa es la razón de que existan muchos y que sea difícil encontrar “el mejor”, ya que eso dependerá de las características particulares del proyecto.

Esa es la razón por la que un arquitecto de sistemas debe conocer lo que existe disponible en el mercado, para que dada una situación pueda elegir los frameworks que mejor resuelvan las problemáticas y determinar lo que ningún framework podrá resolver para desarrollarlo en el proyecto.

Los frameworks suelen enfocarse a resolver problemas de un cierto dominio (área de conocimiento). Estos dominios pueden ser muy diversos, pero están íntimamente ligados al dominio del sistema a construir.

Para un sistema que realice operaciones compra/venta en línea, será importante contar con un framework especializado en realizar este tipo de transacciones, y lo mismo para un sistema bancario.

Por otro lado, existen frameworks que se enfocan en mejorar la “plomería” y arquitectura de las aplicaciones.

Uno de los más conocidos es Hibernate, por su innovadora forma de manejar la persistencia de datos hacia una base de datos, en la que prácticamente el desarrollador puede trabajar con objetos y no con sentencias SQL, agregando así una capa más alta de abstracción. Los frameworks enfocados a este dominio se denominan Mapeo Objeto-Relacional (ORM por sus siglas en inglés Object Relational Mapping).

Otro framework muy utilizado es Spring, enfocado en hacer la “plomería” de sistemas empresariales más sencilla. Está compuesto de varios módulos que incluyen:

- Manejo de transacciones
- Manejo de errores SQL
- Integración con ORM
- Programación Orientada a Aspectos (AOP, Aspect Oriented Programming)
- Inyección de dependencias
- Modelo Vista Controlador (MVC)
- Autenticación y autorización
- Manejo de mensajes

- Así como otros más

El desarrollo de aplicaciones web es un área en la que muchos desarrolladores de sistemas se han enfocado por varias décadas, ya que este modelo cliente-servidor permite que los clientes no requieran nada más que un navegador para poder hacer uso de ella. Ello explica que existan muchos frameworks en el mercado que intentan poner al alcance del programador, herramientas que hacen que el desarrollo sea más sencillo y estructurado.

Un estudio realizado en el 2012, por ZeroTurnaround, Inc., en el cual entrevistaron a 1800 desarrolladores Java, arroja que existen 8 frameworks web de amplio uso en la actualidad:

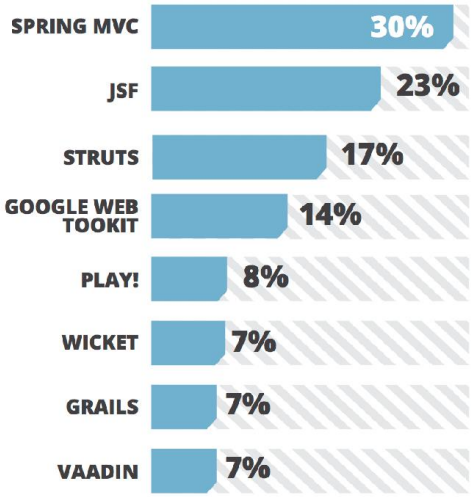

FIGURA II.4 - USO DE FRAMEWORKS JAVA PARA DESARROLLO WEB MÁS POPULARES

En un artículo más reciente (2013) se pretende evaluar y calificar los frameworks en ocho aspectos, a fin de servir de guía al arquitecto del sistema al momento de elegir el que mejor se adapte a las necesidades del proyecto. Y como ellos mismos señalan “Los frameworks web son todos muy diferentes y han sido creados por diferentes razones y para lograr objetivos diferentes”, existen muchas características que pueden influenciar la decisión de usar uno en vez de otro, pero dependerá del tipo de aplicación que se desee construir.

Los ocho aspectos evaluados fueron:

- Creación rápida de aplicaciones prototipo
- Complejidad de framework
- Facilidad de uso
- Documentación y comunidad
- Ecosistema del framework
- Rendimiento y escalabilidad

- Mantenimiento de código y actualizaciones
- Vista y experiencia de usuario

Y los resultados fueron los siguientes:

Framework	Creación Rápida de Prototipos	Complejidad del Framework	Facilidad de Uso	Documentación y Comunidad	Ecosistema del Framework	Rendimiento y Escalabilidad	Mantenimiendo de Código y Actualizaciones	Vista y Experiencia de Usuario
Spring MVC	2.5	3.5	3.0	4.0	4.0	4.0	3.0	2.0
Grails	**5.0**	3.0	**4.5**	**5.0**	**4.5**	4.0	**4.5**	4.0
Vaadin	4.5	**4.0**	**4.5**	**5.0**	3.0	**4.5**	4.0	**5.0**
GWT	4.0	**4.0**	4.0	4.5	3.0	**4.5**	4.0	**5.0**
Wicket	3.5	2.5	3.5	3.0	3.0	3.0	**4.5**	3.5
Play	**5.0**	2.0	3.5	4.0	**4.5**	5.0	4.0	3.0
Struts	2.0	**4.0**	3.0	2.5	3.0	3.0	3.0	2.5
JSF	3.0	3.5	4.0	4.5	4.0	4.0	4.0	4.5

Framework	Puntaje Global
Grails	**34.5**
Vaadin	**34.5**
GWT	33.0
JSF	31.5
Play	31.0
Wicket	26.5
Spring MVC	26.0
Struts	23.0

FIGURA II.5 - RESULTADOS DEL ESTUDIO DE DIVERSOS FRAMEWORKS WEB

No es tema de esta tesis discutir los frameworks en el estudio, ni los resultados que arroja, sólo resaltar que pueden existir muchos frameworks distintos diseñados para resolver una misma problemática general (en este caso, la problemática de desarrollar una aplicación basada en web).

Igualmente hacer notar que las circunstancias de cada proyecto son distintas y los frameworks van cambiando continuamente (puesto que salen nuevas versiones que refinan su funcionamiento o nuevos frameworks que usan otros paradigmas o nuevas tecnologías), por lo que un arquitecto de sistemas bien informado en cuanto a lo que existe disponible podrá tomar mejores decisiones.

También, es cada vez más común que una empresa vea conveniente crear frameworks, para uso interno, que faciliten el desarrollo de algún aspecto de sus sistemas. Una empresa de tecnologías de la información (TI), por ejemplo, puede notar que la creación de cierto tipo de reportes le consume mucho tiempo de desarrollo con las herramientas que ha usado hasta el momento. Si se espera que siga creando este tipo de reportes en el futuro, e incluso para proyectos diferentes, puede decidir construir un framework que agilice este proceso.

Para que la construcción del framework sea viable, se debe estimar el tiempo que se ahorrará en cada reporte construido, y así calcular la cantidad mínima de reportes que se deberán construir para sobrepasar el tiempo y esfuerzo que se estima tomará crearlo en primera instancia.

Muchos frameworks populares y ampliamente usados en sistemas de TI empezaron justamente así, atendiendo una necesidad pequeña, no satisfecha y repetitiva. Con el tiempo estas herramientas fueron agregando características deseables, facilidades de uso, integraron nuevas tecnologías y ampliaron su aplicabilidad a proyectos, aumentado el interés en la herramienta, lo que propicia la formación de comunidades alrededor de la herramienta que impulsan su difusión y uso entre los programadores.

III. PRINCIPIOS DE DISEÑO

INTRODUCCIÓN

Los “principios de diseño de software” son un grupo de directrices que ayudan al programador a no tener un mal diseño en su sistema.

En un sistema, existen tres importantes características que deben ser evadidas (en la medida de lo posible) a fin de evitar un mal diseño:

- **Rigidez**: Cuando es difícil cambiar una parte del sistema, ya que el cambio afecta a muchas otras partes del mismo.
- **Fragilidad**: Cuando al realizar un cambio en una parte del sistema, es muy probable que otras partes del mismo empiezen a funcionar de forma inesperada.
- **Inmovilidad**: Cuando es difícil de reusar en otra aplicación ya que está fuertemente asociado a la aplicación actual.

Todos los principios de diseño son una guía, no una ley, presentan un aspecto deseable (la habilidad de enfrentar algún cambio o permitir la reusabilidad), que sin embargo no siempre aplica a todos los casos.

Si se siguieran al pie de la letra todos los principios de diseño, nunca podríamos tener un código ejecutable, por lo que es inevitable romperlos durante la codificación para que el sistema trabaje. Sin embargo, hay que tener claro que cuando no se emplea un principio, se deben tener claras las consecuencias a fin de estar dispuestos a aceptarlas con tal de tener un código funcional.

PRINCIPIOS DE DISEÑO FUNDAMENTALES

A principios de la década del 2000, Robert C. Martin recolectó cinco principios básicos del diseño orientado a objetos. Cuando estos principios se aplican en conjunto, es más probable que un desarrollador pueda crear un sistema que sea fácil de mantener y enfrentar el cambio a lo largo del tiempo.

- Single responsibility: Principio de responsabilidad única
- Open-closed: Principio abierto cerrado
- Liskov substitution: Principio de sustitución de Liskov
- Interface segregation: Principio de segregación de Interfaces
- Dependency inversion: Principio de inversión de dependencias

Michael Feathers introdujo el acrónimo S.O.L.I.D. (por sus siglas iniciales en inglés) para referirse a estos cinco principios por considerarlos indispensables en el diseño de sistemas robustos.

Con el tiempo se han publicado más principios de diseño en distintos libros, blogs y publicaciones especializadas, entre los más conocidos se encuentran los siguientes:

- Don't Repeat Yourself (DRY): Evita la duplicación
- Composition over inheritance: Composición sobre herencia
- Law of Demeter: Principio del mínimo conocimiento
- Inversion of Control (IoC): Inversión de control (Principio Hollywood)
- Encapsulate what changes: Encapsula el cambio
- Loosely coupled designs: Diseños con bajo acoplamiento

PRINCIPIO DE RESPONSABILIDAD ÚNICA

"Una clase debe tener sólo una razón para cambiar"

Si una clase tiene un solo propósito (responsabilidad), en un futuro, la única razón por la que esa clase debe ser cambiada, es debido a que la forma de realizar ese propósito haya cambiado.

Si una clase tiene más de una responsabilidad, es más susceptible a ser cambiada en el futuro y que al afectar una de sus responsabilidades se pueda afectar (sin proponérselo) a otra de sus múltiples responsabilidades, originando efectos no esperados en el sistema.

Lo que este principio sugiere es que cada clase debe tener solamente un objetivo y si se llegara a detectar que una clase tiene dos o más objetivos, se debería dividir la clase en otras más pequeñas a fin de lograr que cada una de ellas tenga sólo uno.

Por ejemplo, si tenemos una clase Juego definida por el siguiente diagrama:

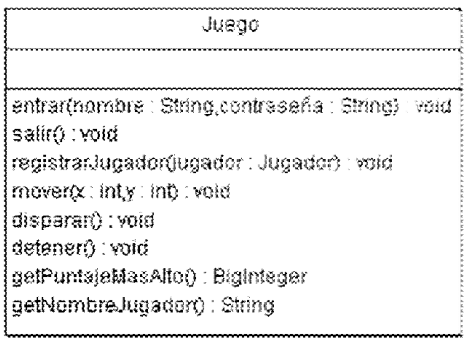

FIGURA III.1 - DIAGRAMA DE CLASE: JUEGO (BAJA COHESIÓN)

Es posible notar que la clase tiene múltiples responsabilidades, entre las que se incluyen el identificar al jugador, permitirle entrar al juego, mantener su puntaje y ejecutar las acciones en el juego (moverse, disparar, detenerse, etc.).

Lo que este principio de diseño sugiere es dividir la clase Juego en múltiples clases de tal forma que se repartan de forma coherente las responsabilidades:

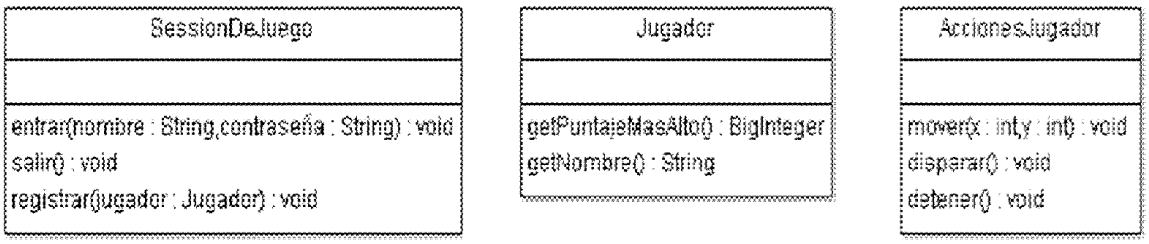

FIGURA III.2 - DIAGRAMA DE CLASE: JUEGO (ALTA COHESIÓN)

PRINCIPIO ABIERTO CERRADO

"Una clase debe estar abierta a poder extenderla, pero cerrada a ser modificada"

El principio dice que la clase debe tener algún mecanismo para extender su funcionalidad de tal forma que permita adaptarse a las necesidades de diferentes sistemas concretos, pero dichas extensiones no deben afectar el correcto funcionamiento de la clase.

Esto se logra con diferentes estrategias, pero todas se basan en usar clases abstractas (interfaces) para describir el comportamiento y clases concretas para implementarlo.

El Template Pattern y el Strategy Pattern son dos patrones de diseño que ocupan este principio para lograr que un algoritmo se pueda reusar en una variedad de situaciones.

Este es un principio genérico que aplica a clases, funciones y módulos.

PRINCIPIO DE SUSTITUCIÓN DE LISKOV

*Sea **q (x)** una propiedad comprobable acerca de los objetos **x** de tipo T. Entonces q (y) debe ser verdad para los objetos **y** del tipo **S** donde **S** es un subtipo de **T**.*

Si la clase S es un subtipo de la clase T, entonces los objetos de tipo T en una aplicación, pueden ser *sustituidos* por objetos de tipo S (i.e. los objetos de tipo S pueden ser *sustitutos* de objetos de tipo T), sin alterar ninguna de las propiedades de esa aplicación.

Entre los conceptos básicos de la orientación a objetos, se enseña la herencia permite crear subclases y con ello pareciera que estás subclases se pueden usar como sustituto de la clase padre en cualquier situación. Si bien esto es cierto, la realidad es que las clases hijas pueden romper fácilmente la funcionalidad de la clase padre al sobrescribir los métodos de éste.

Es en estos casos cuando se dice que una clase hija no cumple con el principio de sustitución, ya que su implementación altera el funcionamiento original de la clase padre.

Un ejemplo comúnmente usado para explicar este fenómeno es un cuadrado que hereda de un rectángulo, puesto que la matemática indica que un cuadrado es simplemente un caso especial de un rectángulo, en el que su ancho y alto miden siempre igual.

Para modelar éste razonamiento es lógico crear una clase `Rectangulo`, capaz de cambiar su ancho y alto, y además agregamos la funcionalidad de poder calcular su propia área.

```
public class Rectangulo {
    private double base;
    private double altura;

    public double getBase() {
        return base;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public double getAltura() {
        return altura;
    }

    public void setAltura(double altura) {
        this.altura = altura;
    }

    public double area() {
        return base * altura;
    }
}
```

Siendo así, si un programador quiere asegurarse de que el cálculo del área se esté realizando correctamente, podría escribir una prueba de unidad como la siguiente:

```
    public static void main(String[] args) {
        Rectangulo rectangulo = getRectangulo();
        pruebaCalculoDeArea(rectangulo);
    }

    private static void pruebaCalculoDeArea(Rectangulo rectangulo) {
        rectangulo.setBase(50);
        rectangulo.setAltura(10);

        System.out.println(rectangulo.area());
        assert rectangulo.area() == 500;
    }

    private static Rectangulo getRectangulo() {
        return new Rectangulo();
    }
```

La prueba le da al rectángulo una base de 50 y una altura de 10, con lo que se espera que el área sea de 500 (50x10). El programador corre esta prueba y todo sale correctamente.

Ahora, si otro programador crea una clase Cuadrado que hereda de Rectangulo y se sobrescriben los métodos `setBase()` y `setAltura()` para agregar las restricciones propias de un cuadrado, la clase podría verse de la siguiente manera:

```
public class Cuadrado extends Rectangulo {
    @Override
    public void setBase(double base) {
        super.setBase(base);
        super.setAltura(base);
    }

    @Override
    public void setAltura(double altura) {
        super.setAltura(altura);
        super.setBase(altura);
    }
}
```

Si el segundo programador sustituye en la prueba del primer programador el `Rectangulo` por un `Cuadrado`, espera que el cálculo del área siga manteniéndose correcto, después de todo, la forma de calcular el área es la misma en un rectángulo y en un cuadrado.

```
    private static Rectangulo getRectangulo() {
        return new Cuadrado();
    }
```

Pero se lleva la sorpresa de que la prueba falla.

```
100.0
Exception in thread "main" java.lang.AssertionError
com.principles.liskov.RectanguloTest.main(RectanguloTest.java:19)
com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

La prueba calcula un área de 100, cuando esperaba un área de 500.

La prueba falló debido a que quien escribió la prueba asumió que es factible modificar la base y la altura de un rectángulo sin ningún tipo de restricción, lo que era verdad al momento de escribir la prueba. Sin embargo la clase hija, `Cuadrado`, sobrescribió esa regla, con lo que rompió el principio de Sustitución.

PRINCIPIO DE LA SEGREGACIÓN DE INTERFACES

"Los clientes no deben ser forzados a depender de interfaces que no usan"

El principio dice que se debe tener cuidado al diseñar una interfaz, ya que debe tener métodos que pertenezcan a la interfaz. Si se añaden métodos que no deberían estar allí, las clases que los implementen tendrán que incluirlos aún si no los usan.

Interfaces que contienen métodos que no les corresponde son llamadas Interfaces “contaminadas” o “gordas” y deben ser evitadas. Una forma común de hacerlo es que al detectar una interfaz contaminada, ésta se divida en varias interfaces no contaminadas.

Por ejemplo, cuando se desarrolla una aplicación gráfica de escritorio con Swing del API estándar de Java y se requiere detectar cuando el usuario quiere cerrar una ventana, el programador debe implementar la interfaz `java.awt.event.WindowListener`.

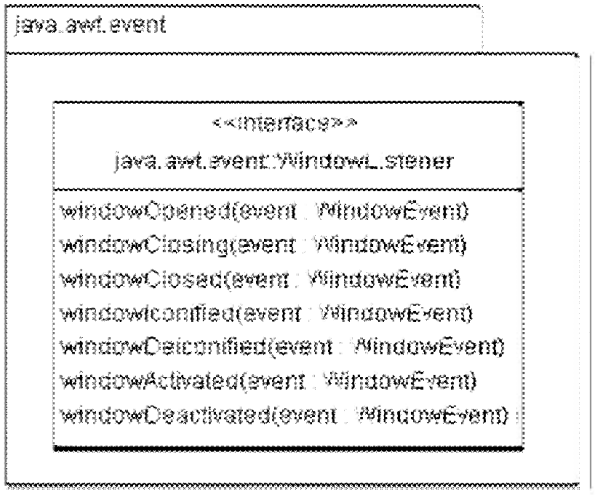

FIGURA III.3 - INTERFAZ WINDOW LISTENER DEL JAVA ESTANDAR API

Sin embargo esta interfaz tiene siete métodos que el programador debe implementar, pero que podrían no ser de su interés en ese momento, por lo que tendría que declarar seis métodos y dejarlos vacíos o lanzar en ellos un “UnsupportedOperationException”.

Lo que este principio señala, es que esa interfaz podría haber sido segregada o dividida en partes más pequeñas que agrupen funcionalidades más específicas.

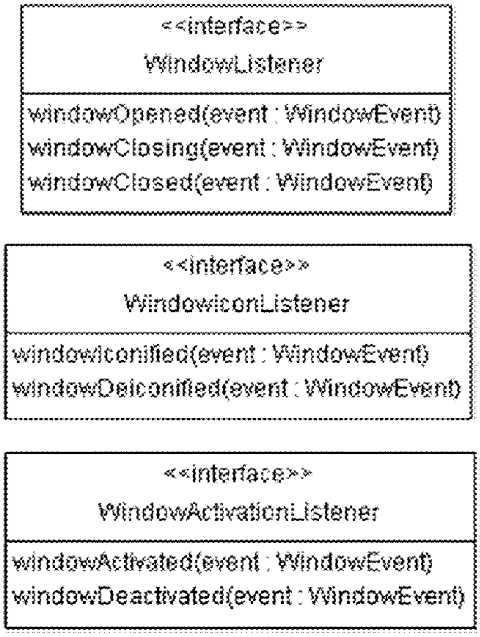

FIGURA III.4 - INTERFAZ WINDOWLISTENER SEGREGADA Y AGRUPADA POR FUNCIONALIDAD

Llevando este principio de diseño al extremo, podríamos también terminar con una interfaz por operación, de forma que nunca haya que implementar algo que no es deseado.

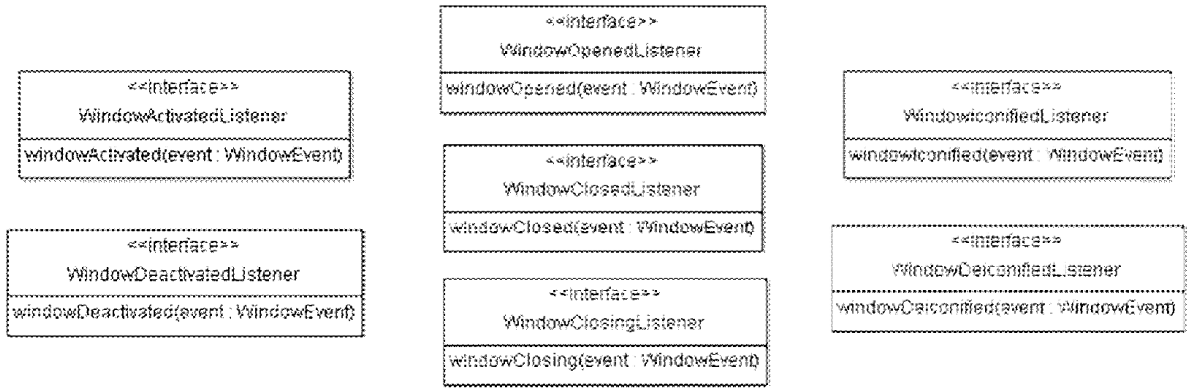

FIGURA III.5 - INTERFAZ WINDOWSLISTENER SEGREGADA EN SU TOTALIDAD

Sin embargo esto hace que crezca el número total de clases y cuando el programador esté interesado en todos los eventos, debe implementar muchas interfaces.

PRINCIPIO DE INVERSIÓN DE DEPENDENCIAS

"Los componentes de 'alto nivel' no deben depender de componentes de 'bajo nivel', en vez de eso, ambos deben depender de abstracciones"

El principio sugiere implícitamente los siguientes puntos:

- Las variables no deberían ser referencias a una clase concreta.
- Las clases no deberían heredar de clases concretas (usar interfaces o clases abstractas).
- Los métodos no deben sobrescribir métodos heredados.

No se puede respetar estos puntos al pie de la letra, puesto que son cosas que permite el lenguaje y hay situaciones en los que es apropiado hacerlo.

Lo que se deriva de este principio es que es preferible trabajar con interfaces/clases abstractas (T) que con implementaciones concretas (S) en el código cliente.
Si una clase hace uso de una interfaz y nunca se entera de la implementación que realmente está usando, entonces el código es mucho más flexible, ya que en un momento dado se podría sustituir la implementación por otra diferente y el código que hace uso de la interfaz no se vería afectado.

Por ejemplo, en el API de Java 5.0 es muy común usar algún tipo de colección para enlistar una serie de objetos.

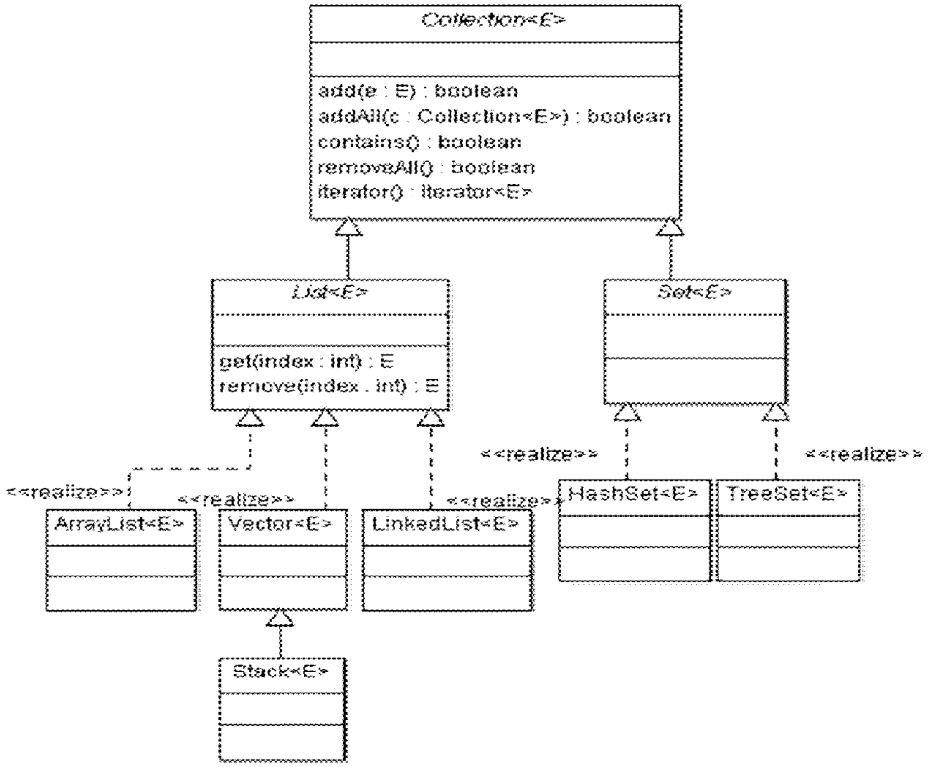

FIGURA III.6 - DIAGRAMA DE CLASES SIMPLIFICADO DE COLECCIONES JAVA 5.0

Lo que el principio de sustitución señala es que si tenemos una referencia a un objeto `ArrayList` de nombre lista:

```
ArrayList<Object> lista = new ArrayList<Object>();
```

Es preferible sustituirla por una clase lo más abstracta posible, como por ejemplo:

```
List<Object> lista = new ArrayList<Object>();
```

O incluso es preferible un nivel aún más abstracto:

```
Collection<Object> lista = new ArrayList<Object>();
```

El nivel de abstracción que se puede alcanzar será determinado por el uso que se dará al objeto, ya que entre más abstracto el objeto, se tendrá acceso a menos métodos propios de la clase concreta.

La clase `ArrayList` tiene algunos métodos útiles como `removeRange(int fromIndex, int toIndex)`, que quitan de la lista a los elementos en un rango determinado, pero fuera de eso, casi todo lo demás se puede hacer con una referencia `List` (principalmente el acceso a un elemento de la lista por su posición en el arreglo con el método `get(int index)`). Sin embargo, si el arreglo se va a acceder siempre de forma secuencial, poder accederlo por índice no es tan útil y sería entonces preferible usar una referencia de tipo `Collection` que contiene un iterador.

La razón de que el principio sugiera usar clases lo más abstractas posible, radica en la flexibilidad que proporcionan para que en un momento dado se pueda realizar un cambio. Por ejemplo si se decidiera que utilizar la clase concreta Vector fuera más apropiado para

un ambiente multihilos por tener sus métodos de acceso sincronizados, entonces lo siguiente no compilaría:

```
ArrayList<Object> lista = new Vector<Object>();
```

Habría que cambiar el tipo de la lista también a `Vector` y no tendríamos la certeza de que eso sería suficiente, ya que si posteriormente se usaban métodos como `removeRange` (o cualquier otro propio de un `ArrayList`) entonces el código seguiría sin poder compilar e implicaría muchas modificaciones para que el código funcione como lo hacía con un `ArrayList`.

Sin embargo, si se usara un `List`, el cambio a un `Vector`, resulta totalmente transparente para el resto del código:

```
List<Object> lista = new Vector<Object>();
```

Y si se usa un Collection se puede hacer lo mismo

```
Collection<Object> lista = new Vector<Object>();
```

Y un poco más, puesto que `Collection` nos permitiría cambiar el objeto por un `Set`, en el caso de que requiriéramos que la lista no contenga duplicados (y el orden no importe):

```
Collection<Object> lista = new HashSet<Object>();
```

O si el orden es importante, pero no queremos que contenga duplicados, podemos usar un `LinkedHashSet`:

```
Collection<Object> lista = new LinkedHashSet<Object>();
```

Si en una aplicación, para una clase `Persona`, se requiere guardar un listado de sus hijos (también de tipo `Persona`), el principio indica que resulta preferible usar una clase de alto nivel (abstracta) para la colección, que una de bajo nivel (concreta), a fin de que el código que utilice el método `getHijos()` no se entere de la implementación concreta que se está utilizando para ordenar la colección.

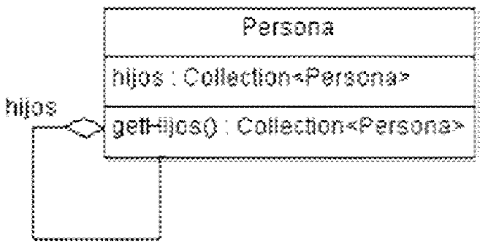

FIGURA III.7 - DIAGRAMA DE CLASES: PERSONA

EVITA LA DUPLICACIÓN

"Cada pieza de conocimiento debe tener una única, no ambigua y autorizada representación dentro de un sistema"

La duplicación de "piezas de conocimiento" en un sistema puede conducir a dificultar el mantenimiento, y fácilmente producir contradicciones, que pueden confundir a los programadores y llevar a incongruencias en una aplicación.

Desde que estudiamos un lenguaje de programación, aprendemos con ejemplos concretos, que luego copiamos y modificamos según veamos conveniente. Entre los programadores es muy difundida la técnica de copiar y pegar código fuente, que al hacerle unas "pequeñas" modificaciones se adaptará a la necesidad a la que le hagan frente. Si bien esta es una técnica muy didáctica y fácil de aplicar, es necesario cuidarnos de no repetir lógica y datos, ya que estos se traducirán en un mayor número de líneas de código, más complejo, dificultando su lectura y proporciona un mayor número de oportunidades de caer en incongruencias y producir resultados no deseados.

Lo que este principio señala es que se debe evitar al máximo la duplicación, en cuanto se detecta que existe una "parte" muy similar a otra, se debe hacer lo posible por refactorizarlo, generalizando el código común y separando las diferencias, de tal forma que se elimine el código duplicado.

Aunque muchas veces este principio se centra en **nunca** duplicar código fuente, sus autores Andy Hunt y Dave Thomas en su libro "The Pragmatic Programmer", amplían este concepto a todos los aspectos de un sistema, incluyendo esquemas de base de datos, modelos, planes de pruebas e incluso la documentación.

Precisan que no se trata de nunca tener duplicados, sino que más bien se trata de que debe existir una única fuente donde se deben realizar los cambios y que mediante procesos automatizados se propaguen los cambios para tener el sistema en un estado consistente.

COMPOSICIÓN SOBRE HERENCIA

"Favorece la Composición sobre la Herencia"

La herencia es un recurso para reutilizar código al permitir a las clases hijas heredar funcionalidad del padre, extenderla y modificar el comportamiento original al permitir sobrescribir los métodos heredados.

Sin embargo, una desventaja de este esquema es que puede complicar el mantenimiento del sistema si los cambios afectan a la clase padre, ya que al agregar, quitar o modificar métodos heredados se puede afectar, sin proponérselo, a las clases hijas. Se pueden crear comportamientos no esperados en el sistema, tener que revisar y modificar, en su caso, cada una de clases hijas.

Igualmente, si una clase hija realiza una implementación que otras clases hermanas requieren, no es posible compartirlo, haciendo necesario "copiar y pegar" la

implementación en cada una de las clases que lo requieran, provocando duplicidad en el código, complicando el mantenimiento del sistema.

Otra desventaja es que, en lenguajes que no soportan la herencia múltiple, una clase no puede heredar el comportamiento que requiere de dos padres distintos. Y en aquellos lenguajes donde se permite la herencia múltiple se aumenta la complejidad y la ambigüedad del resultado.

Existe otra técnica conocida como “composición”, que permite a un objeto reutilizar el código de otro objeto a través de una referencia al segundo objeto y una invocación a la funcionalidad requerida.

La composición permite un diseño más flexible ya que permite a un objeto invocar la funcionalidad de otros objetos sin tener que heredar de ellos, también tiende a separar de mejor forma las responsabilidades de cada clase, permite agregar nueva funcionalidad agregando código nuevo sin alterar código existente y se puede cambiar la implementación en tiempo de ejecución.

Por ejemplo, si se tiene previamente una clase `Motor`, con sus útiles métodos enciende y apaga, y se desea ahora desarrollar la clase `Automovil`, que como parte de su proceso de encendido requiere de encender un motor, se podría recurrir a la herencia para que de esa forma la clase `Automovil` incluya implícitamente los métodos `enciende()` y `apaga()` de la clase `Motor`:

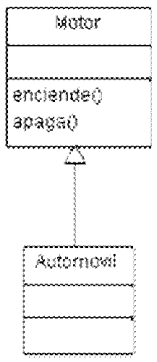

FIGURA III.8 - DIAGRAMA DE CLASES: AUTOMOVIL/MOTOR CON HERENCIA

En el lenguaje Java, esto permitiría crear una instancia de Automovil, e invocar sobre éste los métodos heredados:

```
Automovil automovil = new Automovil();
automovil.enciende();
```

Sin embargo, lo que este principio señala es que antes de recurrir a la herencia, se debe ver si la composición no es más apropiada para este caso:

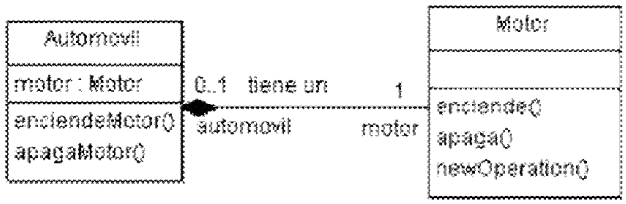

FIGURA III.9 - DIAGRAMA DE CLASES AUTOMOVIL/MOTOR CON COMPOSICIÓN

En el diagrama de clases se muestra a un `Automovil` con un atributo `motor`, lo que se puede leer como "Un automovil tiene un motor" y define dos métodos nuevos `enciendeMotor()` y `apagaMotor()`.

Estos nuevos métodos son implementados de tal forma que delegan la funcionalidad a la clase Motor:

```
public void enciendeMotor() {

    motor.enciende()

}
```

Y entonces su uso es de la siguiente forma:

```
Automovil automovil = new Automovil(new Motor());
automovil.enciendeMotor();
```

Este diseño es más flexible que el anterior pues permite al automóvil cambiar, incluso en tiempo de ejecución, la implementación de su motor (por ejemplo, si existieran las clases `MotorDiesel` y `MotorGasolina`, que heredan de `Motor`) y permite también que el `Automovil` sea una composición de varias otras clases, como puede ser un `Acumulador`, unas `Llantas` o unas `Puertas`.

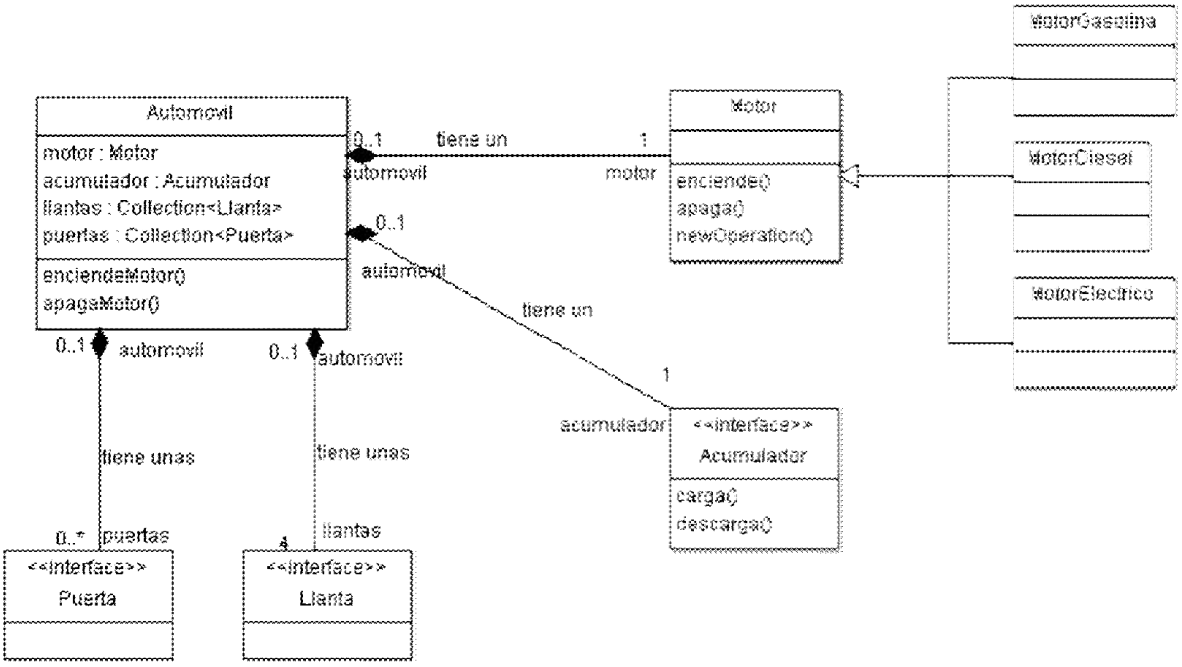

FIGURA III.10 - DIAGRAMA DE CLASES AUTOMOVIL Y LAS PARTES QUE LO COMPONEN

El principio señala que "tiene un" (Composición) es mejor que "es un" (Herencia), pero donde tenga sentido utilizar la herencia, ésta se debe usar.

PRINCIPIO DEL MÍNIMO CONOCIMIENTO

"Háblale solamente a tus amigos más inmediatos"

Un objeto debe reducir al mínimo la cantidad de objetos con los que interactúa directamente, ya que esto incrementa las dependencias del objeto con otros, causado un alto acoplamiento.

En un sistema con alto acoplamiento los cambios en una parte, potencialmente afectan a mucha otras partes, haciendo al sistema frágil, difícil de mantener y entender.

El principio sugiere que toda operación de un objeto sólo tendría que utilizar:

- Las operaciones propias del objeto.
- Los objetos que tenga asociados o sean atributos del objeto.
- Los objetos que recibe como parámetro la operación.
- Los objetos que cree la propia operación.

Regresando al ejemplo del automóvil con un motor (véase Figura III.9), lo que este principio señala es que se deben evitar las sentencias como la siguiente:

```
Automovil auto = new Automovil();
auto.getMotor().enciende()
```

Ya que hace que el código que lo usa dependa no solamente del objeto auto que se ha creado ahí (por lo que no viola el principio), pero también depende del objeto Motor y de las operaciones que éste define.

El principio sugiere que `Automovil` debería tener una operación `enciendeMotor()` que encapsule la invocación a `Motor`, para que de esta manera, el código cliente ya no sea dependiente directo de la clase `Motor`.

```
auto.enciendeMotor();
```

PRINCIPIO DE INVERSIÓN DE CONTROL

"No nos llames, nosotros te llamamos"

Tradicionalmente, el código de una aplicación particular invoca rutinas de librerías reutilizables, por lo que se dice que mantiene el control del flujo del programa.

Lo que el también llamado "Principio Hollywood" sugiere, es que en algunas ocasiones puede ser mejor invertir la situación y que sea la librería reutilizable la que tome el control del flujo y decida el momento apropiado para invocar el código particular de la aplicación.

Este principio es la piedra angular de los frameworks y es la razón por lo que suelen ser más intrusivos que una librería de rutinas tradicional. Se sacrifica el control, pero se

consigue reutilizar, no sólo rutinas simples, sino también flujos de control complejos, en donde el programador puede tener que introducir la lógica de su aplicación particular en lugares y de formas muy concretas, definidas por el framework.

Esto restringe, y al mismo tiempo libera, al programador de tener que tomar ciertas decisiones de diseño, lo que promueve el orden y la homogeneidad del código de la aplicación.

Existen varias formas de lograr esto, pero lo básico es que los componentes de más bajo nivel (subclases) se "enganchen" a clases de más alto nivel (interfaces), proporcionando implementaciones, que el framework no sabe exactamente qué hacen, pero sí decide cuándo ejecutarlas.

Patrones de diseño como el Factory, el Template Method y el Strategy se basan en este principio.

ENCAPSULA EL CAMBIO

"Identifica los aspectos que varían y sepáralos de lo que se mantiene constante"

Un aspecto que varía debería ser representado por una interfaz, de tal forma que se encapsule ese aspecto, se pueda separar del resto y pueda haber varias implementaciones de él.

En una aplicación suele haber cosas que por su naturaleza son propensas a cambiar y otras cosas que son relativamente estáticas. Durante el análisis, es necesario identificar las que pueden variar para procurar aislarlas del resto, y así poder realizar los cambios que se vayan presentando sin tener demasiadas consecuencias sobre el resto del sistema. Para lograr este encapsulamiento hay una variedad de técnicas propuestas por la mayoría de los patrones de diseño (ej. Strategy Pattern), pero es fundamental que el diseñador pueda identificar los aspectos que pueden variar en una aplicación en particular.

DISEÑOS CON BAJO ACOPLAMIENTO

"Busca diseños de acoplamiento débil entre objetos que interactúan"

Un objeto debe poder comunicarse con otros objetos para realizar cualquier trabajo no trivial, sin embargo saber demasiado de los objetos con los que interactúa provoca que sean más frágiles y difícil separarlos, complicando el poder reusar alguna parte del sistema y provocando que cualquier cambio en uno, pueda afectar al otro, dificultando así el mantenimiento del sistema.

Lo deseable es que un objeto conozca lo menos posible de los objetos con los que interactúa. Debe limitarse a tener una forma de enviar mensajes a otros objetos evitando conocer detalles que no son de su competencia.

A este grado de dependencia entre objetos se le denomina "acoplamiento", por lo que un buen diseño debe procurar un bajo acoplamiento.

IV. PATRONES DE DISEÑO

HISTORIA

A finales de los años setentas, el arquitecto Christopher Alexander, junto con varios colegas suyos, publicó “A Pattern Language: Towns, Buildings, Construction”, un libro que pretende formalizar y plasmar de una forma práctica generaciones de conocimiento arquitectónico. Esto lo hacen presentando una serie de 253 patrones para la construcción de ciudades/edificios/hogares, que colectivamente forman lo que los autores llamaron un “lenguaje de patrones”.

En palabras de los autores "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez".

“Un patrón define una posible solución correcta para un problema de diseño dentro de un contexto dado, describiendo las cualidades invariantes de todas las soluciones”, y esto parece aplicar a todo tipo de problemas de arquitectura desde diseñar una ciudad, hasta dónde deben ir las perillas de las puertas.

Unos años después, Kent Beck y Ward Cunningham empezaron a extrapolar el concepto de patrones de diseño fuera del ámbito de la construcción y estudiaron su aplicabilidad en el diseño de software. En 1987 usaron cinco patrones para asistir a una empresa con problemas para terminar el diseño de un proyecto de software. Estos patrones capturaban las buenas ideas de arquitectos de software experimentados para luego poder comunicar a programadores menos experimentados la forma de resolver un problema en particular. Estos patrones fueron publicados en un artículo con el título “Using Pattern Languages for OO Program**”.**

Mientras tanto, Erich Gamma también se daba cuenta de la importancia de los patrones de diseño recurrentes mientras trabajaba en su tesis de posdoctorado. Creía en que los patrones de diseño podrían facilitar la tarea de escribir software orientado a objetos que fuera reutilizable y definió la forma de documentarlos y comunicarlos de forma eficiente.

Junto con Richard Helm, empezó a catalogar patrones de diseño y en 1991 se les unieron Ralph Johnson y John Vlissides, que en total, recopilaron 23 patrones de diseño agrupados en 3 categorías y publicaron “*Design Patterns: Elements of Reusable Object-Oriented Software*” que ha sido el libro de referencia sobre patrones de diseño en el diseño de software, y los autores adquirieron el sobrenombre de la “Banda de los Cuatro” (GoF, Gang of Four).

Los patrones de diseño han continuado su evolución desde la publicación del libro original de GoF, impulsados sobre todo por los retos que los desarrolladores de software han enfrentado con los cambios en hardware y requerimientos en las aplicaciones, se han ido descubriendo y documentando nuevos patrones aplicables a muy diversos ambientes de desarrollo.

DEFINICIÓN

Existen muchas definiciones de lo que es un patrón. En la literatura de patrones de diseño, cada autor intenta definir con más claridad o exactitud técnica el concepto.

La siguiente definición, tomada del libro “Head First Design Patterns” es probablemente la definición más concisa existente:

“Un Patrón es una solución a un problema en un contexto”

Es clara y sobretodo corta, sin embargo es útil desarrollarla un poco más para entender lo que la compone:

- **Contexto**: Es una situación recurrente en la que se aplica el patrón.
- **Problema**: Es el objetivo que se quiere alcanzar dentro del contexto (incluyendo las restricciones que existen en el contexto).
- **Solución**: Un diseño general que resuelve el problema y que cualquiera puede aplicar.

Conocer las bases de la orientación a objetos es fundamental, pero eso no convierte al programador en un buen diseñador de sistemas orientados a objetos, los buenos diseños orientados a objetos son reusables, extensibles y de fácil mantenimiento.

En principio, un sistema se debe realizar con un buen diseño orientado a objetos y si el sistema nunca fuera a cambiar, eso sería suficiente para tener un diseño exitoso.

Sin embargo, esto no suele suceder en un sistema real, ya que el cambio es parte de la naturaleza del propio sistema, por lo que sólo queda lidiar con él de la mejor manera posible (i.e. haciendo que el diseño esté preparado para "administrar el cambio").

Muchas veces la esencia de esos cambios es la misma en una gran cantidad de sistemas y es para ellos que se han descubierto patrones de diseño que aminoran el impacto del cambio.

Estos patrones, son soluciones generales, bien documentadas, conocidas (lenguaje común), probadas por el tiempo y muchos desarrolladores, y que se pueden aplicar a los casos particulares que se presenten al desarrollar un sistema.

Los patrones de diseño muestran la forma de construir sistemas que cumplen con algunas de esas características, no proporcionan código, sino que proporcionan soluciones generales a problemas recurrentes y es cuestión del desarrollador implementarlo de forma específica para la aplicación. La mayoría de ellos intentan resolver problemas derivados del cambio en los sistemas y permiten que una parte del sistema cambie sin afectar otras partes del mismo.

Además, los patrones de diseño proporcionan un lenguaje común entre los desarrolladores, facilitando la comunicación. Es un lenguaje para entender y expresar arquitecturas (cualidades, características y restricciones).

Permite mantener la discusión a nivel del diseño de la arquitectura, ignorando los detalles de la implementación (ya que se sabe que el patrón funciona).

En el desarrollo de un sistema, no todo se debe o se puede resolver con patrones de diseño, ya que los patrones de diseño tienes sus desventajas, por lo que en general se debe optar por la solución más sencilla.

Desventajas de usar patrones:

- Agregan complejidad al diseño (Se agregan clases y objetos)
- Agregan capas (que pueden hacer más ineficiente la comunicación)
- Requiere que las demás personas conozcan los patrones
- Limita el desarrollo de nuevas soluciones creativas que sean diferentes a los patrones conocidos.

Un patrón generalmente agrega un nivel de complejidad al diseño, pero con ello, pretende resolver algún problema de flexibilidad para que no sea tan doloroso realizar algún cambio (esperado) en el sistema.

Los patrones de diseño son una herramienta poderosa, pero no se debe usar indiscriminadamente, siempre se debe optar por la solución más sencilla que resuelva el problema. Es cuando una solución sencilla no satisface los requerimientos de cambio en el sistema, que se busca un patrón que permita realizar los cambios esperados sin afectar al resto del sistema.

El considerar el problema junto con sus restricciones bajo las cuales debe operar la solución, nos llevará de forma natural al patrón a utilizar (si éste existe o es conocido).

Si no existe un patrón que cumpla con el requisito y las restricciones exactamente, se pueden buscar patrones que parezcan resolverlo en parte. Estudiando las ventajas y consecuencias de cada uno, se puede elegir el más apropiado a la situación.

Incluso se puede modificar un patrón conocido si de esa manera resuelve mejor el problema particular, pudiendo dar lugar al descubrimiento de un patrón derivado o incluso un patrón totalmente nuevo para el diseñador.

CLASIFICACIÓN

Cuando la cantidad de patrones crece, se hace necesario clasificarlos de alguna manera. Existen varias maneras de clasificar patrones y muchas veces parece que un patrón podría pertenecer a más de un grupo.

Una clasificación de patrones debe ser de ayuda para comparar patrones y para tener un nivel más de abstracción al resolver problemas, por ejemplo puedo hablar de que requiero un patrón para crear objetos (patrón Creacional), sin referirme específicamente a alguno.

CATÁLOGOS DE PATRONES

El catálogo de patrones más conocido es el realizado por los autores Gamma, Helm, Johnson y Vlissides, comúnmente referidos por el nombre de “La banda de los cuatro” (en inglés GoF: Gang of Four) descrito en el documento “Design Patterns” (Addison Wesley, 1995).

El GoF describe en este documento 23 patrones fundamentales, agrupados, según la problemática que pretendan resolver, en tres grupos:

- Creacionales - Involucrados en la instanciación de objetos. Proveen una forma de desacoplar el cliente de los objetos que requiere instanciar. Ejemplos de estos patrones incluyen Fabricas (Factories) y Constructores (Builders).
- Estructurales - Permiten realizar composiciones de clases u objetos en estructuras más grandes. Ejemplos de estos patrones incluyen Proxy y el Adaptador (Adapter).
- De Comportamiento - Concierne la forma en que clases y objetos interactúan y distribuyen responsabilidad. Ejemplos de estos patrones Comando (Command), Método Plantilla (Template Method), Estado (State), Estrategia (Strategy), Cadena de Responsabilidad (Chain of Responsibility), Observador (Observer), Despachador Múltiple (Multiple Dispatching) y Visitador (Visitor).

Para documentar un patrón e incluirlo dentro de este catálogo, es necesario incluir cierta información que lo describa y distinga de los demás patrones.

- Nombre - Identifica el patrón y ayuda a crear un vocabulario común con otros desarrolladores.
- Intención - Una corta descripción de lo que el patrón hace.
- Clasificación - La categoría en la que recae el patrón.
- Motivación - Describe el problema y la forma en que el patrón lo soluciona.
- Aplicabilidad - Describe situaciones en las que el patrón puede ser aplicado.
- Estructura - Provee un diagrama ilustrando las relaciones entre las clases que participan en el patrón.
- Participantes - Son las clases y objetos del diseño. Se describen sus respectivas responsabilidades y roles dentro del patrón.
- Colaboraciones - Describe la forma en que los participantes interactúan entre ellos dentro del patrón.
- Consecuencias - Describen los efectos que este patrón puede tener (positivos y/o negativos).
- Implementación - Provee técnicas necesarias para implementar el patrón y puntos finos con los que haya que tener cuidado.
- Código de Ejemplo - Provee fragmentos de código que puede ayudar a realizar la implementación.
- Usos Conocidos - Describe ejemplos de sistemas reales donde el patrón es usado.
- Patrones Relacionados - Describe la forma en que el patrón se relaciona con otros patrones conocidos (similitudes, diferencias, posibles combinaciones, etc.).

Existen otras Clasificaciones no incluidas por GoF, como es clasificarlos en:

- Patrones de Clase - Describen relaciones entre clases a través de la herencia. Estas relaciones son establecidas en tiempo de compilación.
- Patrones de Objeto - Describen relaciones entre objetos y son principalmente definidos por composición. Estas relaciones son creadas en tiempo de ejecución, por lo que son más dinámicas y flexibles.

También se han creado más grupos de clasificación, como el de Patrones de Concurrencia, que propone Mark Grand en su libro "*Patterns in Java"*, que agrupa patrones para resolver los problemas que surgen cuando múltiples hilos ejecutan al mismo tiempo código compartido y afectan datos comunes.

Con el tiempo, se han ido descubriendo más patrones, más grupos de clasificación e incluso otras formas de clasificarlos, pero los patrones propuestos por GoF siguen siendo una de las referencias más importantes para los que estudian el tema.

PATRONES FUNDAMENTALES

SINGLETON

Problema: En una aplicación suelen existir ciertos objetos de los que es importante que exista en todo momento una y sólo una instancia del mismo, ya que estos objetos suelen administrar de forma centralizada un recurso y que si llegara a existir más de uno, dicha administración no se realizaría de forma apropiada.

Definición: Este patrón asegura que una clase tenga una y sólo una instancia, y provee un punto de acceso global para accederla.

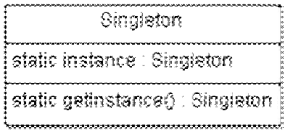

FIGURA IV.1 - DIAGRAMA DE CLASES: SINGLETON

La forma más común de implementar una clase Singleton es proporcionando a esa clase una propiedad privada y estática (`instance`) que almacenará la referencia a la única instancia de la clase.

Para asegurarse que ningún otro código pueda crear una instancia nueva de la clase, se hace que el constructor sea privado y se proporciona un método estático (`getInstance()`) que permite acceder a la propiedad `instance` para obtener la referencia al objeto.

A pesar de ser un patrón de diseño sencillo, su implementación puede ser delicada en ambientes con múltiples hilos de ejecución, ya que si dos hilos intentan crear la instancia al mismo tiempo, sólo uno de ellos debe lograr crear el objeto.

Existen tres técnicas alternativas para implementar un Singleton en un ambiente multihilos:

- Tradicional: Método getInstance sincronizado
- Simple: Dejar la instanciación a cargo de la JVM
- Doble Revisión: Dos validaciones revisan que sea necesario crear una nueva instancia, sólo la segunda es sincronizada.

// Singleton tradicional

```
public class Singleton {
    private static Singleton instance;

    // Constructor privado para no permitir intanciar en cualquier lugar
    private Singleton() {  }

    public static synchronized Singleton getInstance() {
        if (null == instance) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

La forma tradicional tiene la ventaja de postergar la creación del objeto hasta que se requiere usar (cuando se manda llamar el método `getInstance()`). Sin embargo, puesto que dos o más hilos de ejecución podrían mandar llamar este método al mismo tiempo se requiere que el método sea sincronizado y esto agrega un tiempo de respuesta más lento que en algunas aplicaciones puede llegar a ser crítico (puede ser hasta 100 veces más lento el performance de ese método).

// Singleton Simple

```
public class Singleton {
    private static final Singleton instance = new Singleton();

    // Constructor privado para no permitir intanciar en cualquier lugar
    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

En el método simple, es la JVM la que instancía de forma automática el objeto Singleton, ya que lo hace al cargar la clase, por lo que lo hace una sola vez. Por su simplicidad, este método es muy usado, pero no permite al programador definir el momento de instanciar el objeto.

// Singleton de doble revisión (double checking)

```
public class Singleton {
    private static Singleton instance = null;

    // Constructor privado para no permitir intanciar en cualquier lugar
    private Singleton() {}

    // Creador sincronizado para protegerse de posibles problemas  multi-
hilo
    private synchronized static void createInstance() {
        // Se vuelve a revisar, para el caso en que 2 hilos hayan entrado
a la validación anterior al mismo tiempo.
        if (instance == null) {
            instance = new Singleton();
        }
    }

    public static Singleton getInstance() {
        if (instance == null) {
            createInstance();
        }
        return instance;
    }
}
```

Para un Singleton, el único momento potencialmente “peligroso” en un ambiente multihilo es en la creación de la instancia, puesto que debemos asegurarnos que si dos hilos se ejecutan simultáneamente, sólo uno debe crear la instancia, mientras el segundo debe esperar a que esté creado para usarlo.

La técnica “Double Checking” elimina la necesidad de sincronizar el método getInstance(), ya que delega la creación del singleton al método createInstance(). Una vez creada la instancia del singleton, ya no será necesario llamar al método createInstance(), por lo que se eliminará el aumento en el tiempo de respuesta causado por la sincronización.

Aunque esta técnica requiere más líneas de código, es probablemente la forma más elegante, eficiente y entendible de hacerlo, ya que permite postergar la creación de la instancia hasta el momento en que se necesite usar, pero al no estar sincronizado el método getInstance() no causa un aumento en el tiempo de respuesta.

MÉTODO SIMPLE DE FABRICACIÓN

Problema: Al usar la palabra reservada "new" implica que estamos creando una instancia concreta y eso contradice el principio de sustitución de Liskov. Sin embargo, no se podría tener un programa orientado a objetos, si no se instancian objetos en algún momento y no hay otra forma de hacerlo.

Lo que el principio dice es que se debe procurar no instanciar objetos nuevos en todos lados, sino delegar (y encapsular) la creación a un componente particular llamado “Fábrica de Objetos”.

Definición: El “Método Simple de Fabricación” no es propiamente un patrón de diseño, pero es muy usado por su facilidad de implementación.

Básicamente, encapsula en un método la creación de objetos de una clase, escondiendo dentro del método la instanciación concreta.

Ejemplo:

Si existe una clase abstracta `Animal`:

```
    abstract class Animal {
        abstract void hazSonido();
    }
```

Y varias implementaciones concretas que heredan de ésta:

```
    class Vaca extends Animal {
        @Override
        void hazSonido() {
            System.out.println("Muuu");
        }
    }

    class Gato extends Animal {
        @Override
        void hazSonido() {
            System.out.println("Miauu");
        }
    }

    class Perro extends Animal {
        @Override
        void hazSonido() {
            System.out.println("Guau Guau");
        }
    }
```

Suponiendo entonces que el código cliente debe instanciar una de estas clases dependiendo del valor de un parámetro proporcionado en tiempo de ejecución, se podría implementar de la siguiente manera:

```
public class PruebaAnimal {
    public static void main(String[] args) {
        String tipoAnimal = args[0];

        Animal animal = null;

        if(tipoAnimal.equals("vaca")) {
            animal = new Vaca();
        } else if(tipoAnimal.equals("perro")) {
            animal = new Perro();
        } else if(tipoAnimal.equals("gato")) {
            animal = new Gato();
        }

        animal.hazSonido();
    }
}
```

Lo que el “Método Simple de Fabricación” sugiere es encapsular en un solo método la creación de objetos, de tal forma que se oculta del código cliente la instanciación.

Es posible implementar esto creando un objeto que contendrá este método para crear instancias de tipo Animal:

```
    class FabricaAnimal {
        public Animal creaAnimal(String tipoAnimal) {
            if(tipoAnimal.equals("vaca")) {
                return new Vaca();
            } else if(tipoAnimal.equals("perro")) {
                return new Perro();
            } else if(tipoAnimal.equals("gato")) {
                return new Gato();
            }

            throw new RuntimeException(
                  "Tipo de Animal desconocido: " + tipoAnimal);
        }
    }
```

De esta manera se simplifica el código cliente y se tiene en un solo lugar la creación de objetos ayudando a simplificar el mantenimiento y la legibilidad, sobre todo cuando el objeto en cuestión es complicado de construir y/o requiere acceso a información o recursos que no deben estar contenidos en el objeto mismo.

```
public class PruebaAnimal {
    public static void main(String[] args) {
        String tipoAnimal = args[0];
        FabricaAnimal fabrica = new FabricaAnimal();
        Animal animal = fabrica.creaAnimal(tipoAnimal);
        animal.hazSonido();
    }
}
```

Así entonces, si se ejecuta este código con un argumento “perro” se obtiene:

```
>java PruebaAnimal perro
Guau Guau
```

Pero si se ejecuta con un argumento “vaca” se obtiene:

```
>java PruebaAnimal vaca
Muuu
```

FACTORY METHOD

Problema: Al igual que el “Método Simple de Fabricación” este patrón, y todos los patrones de diseño “Fábrica”, encapsulan la creación de objetos, promoviendo así el bajo acoplamiento, al reducir la dependencia de la aplicación hacia clases concretas, al tiempo que evita duplicar el código de instanciación y por tanto facilita el mantenimiento.

Definición: El patrón “Factory Method” define una interfaz para crear un objeto, pero permite a las subclases decidir la clase concreta a instanciar.

A simple vista este patrón se parece mucho al “Método Simple de Fabricación”, sin embargo, a diferencia de éste, el patrón depende de la herencia para realizar su función, con lo que logra que si en un momento dado se requiere “fabricar” otros objetos, se crearán fábricas concretas nuevas, permitiendo esto extender la funcionalidad de la fábrica abstracta sin modificar el código ya existente de las fábricas anteriores, evitando así introducir defectos en el código que ya está funcionado.

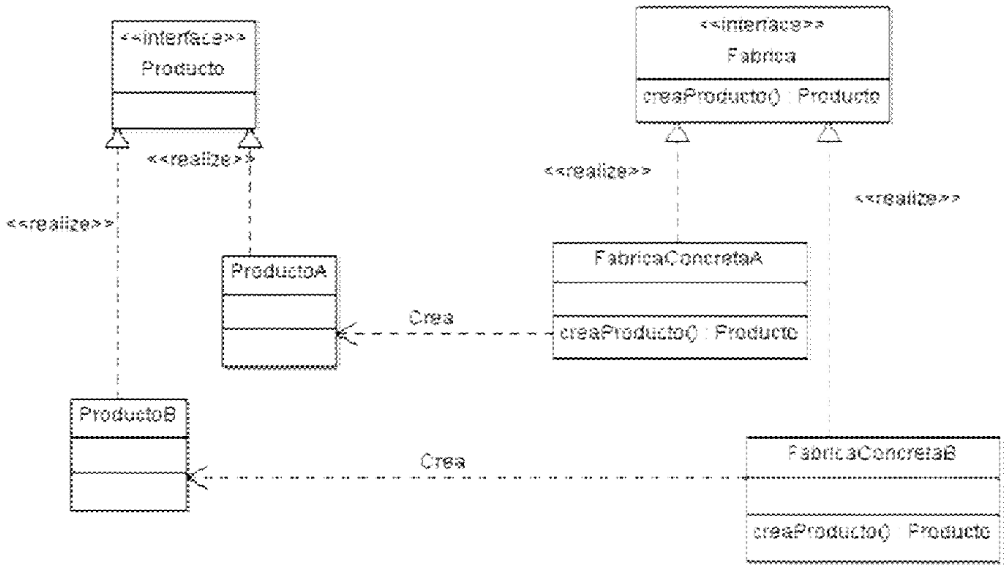

FIGURA IV.2 – DIAGRAMA DE CLASES: FACTORY METHOD

Producto es una interfaz que define a todos los posibles objetos que puede instanciar una `Fabrica`.

`ProductoA` y `ProductoB` son las clases concretas que se desean instanciar y que heredan de *Producto*.

`Fabrica` es la interfaz que define un método `creaProducto()`, cuyo propósito es crear un objeto de tipo `Producto`.

`FabricaConcretaA` y `FabricaConcretaB` son las clases que realmente “deciden” que clase concreta de *Producto* deben instanciar.

El código cliente sólo debe definir qué fábrica concreta debe usar. Por ejemplo, si opta por la `FabricaConcretaA`, obtendrá (sin saberlo) objetos de tipo `ProductoA`, pero siempre los manejará como de tipo `Producto`.

Implementar el patrón descrito en código java, mostraría un código similar al siguiente:

Se crea la interfaz que define a toda la familia de productos que la fábrica pueda crear:

```
public interface Producto {
    String operacion();
}
```

Se definen algunos productos concretos:

```
public class ProductoA implements Producto {
    @Override
    public String operacion() {
        return "Operacion del Producto tipo A";
    }
}
public class ProductoB implements Producto {
    @Override
    public String operacion() {
        return "Operacion del Producto tipo B";

    }
}
```

Se define la interfaz de las Fábricas creadoras de Productos:

```
public interface Fabrica {
    Producto creaProducto();
}
```

Y se definen un par de fábricas concretas.

La *FabricaConcretaA*, creará exclusivamente productos de tipo A (*ProductoA*)

```
public class FabricaConcretaA implements Fabrica {
    @Override
    public Producto creaProducto() {
        return new ProductoA();
    }
}
```

La *FabricaConcretaB*, creará exclusivamente productos de tipo B (*ProductoB*)

```
public class FabricaConcretaB implements Fabrica {
    @Override
    public Producto creaProducto() {
        return new ProductoB();
    }
}
```

De esta manera, cuando el código cliente utiliza un objeto *Producto*, invocando sobre el los métodos que tenga definidos, no tiene necesidad de saber si es de tipo A o de tipo B, simplemente los obtiene de la fábrica adecuada.

```
public class FactoryMethodTest {
    public static void main(String[] args) {
        Fabrica fabrica = new FabricaConcretaA();

        Producto producto = fabrica.creaProducto();

        System.out.println(producto.operacion());
    }
}
```

En este ejemplo, se usa una *FabricaConcretaA*, que sabemos crea productos tipo A, por lo que al invocar el método `operacion()`, por lo que la salida de esta prueba será: `Operacion del Producto tipo A`

Modificación al Patrón: Parametrized Factory Method

En algunas circunstancias se desea que un mismo método fábrica, pueda instanciar una de varias clases concretas de la interfaz *Producto*. Para ello, se puede emplear un parámetro en el método que indique de alguna forma el tipo de objeto a instanciar.

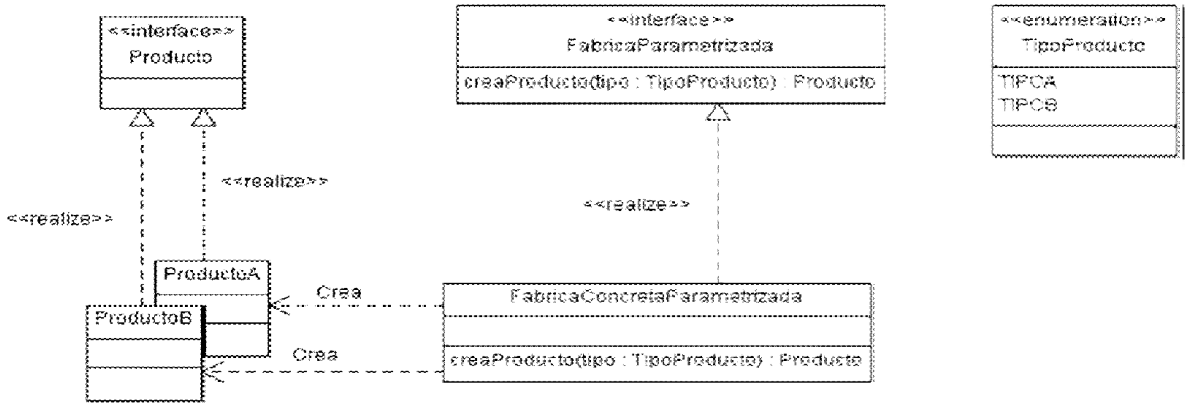

FIGURA IV.3 - DIAGRAMA DE CLASES: FACTORY METHOD PARAMETRIZADO

Por ejemplo, si el código cliente indica lo siguiente:

```
Producto producto = fabrica.creaProducto(TipoProducto.TIPOB);
```

Obtendrá, de forma implícita, un objeto de tipo *ProductoB*.

ABSTRACT FACTORY

Problema: Existen situaciones donde es conveniente crear toda una familia de objetos relacionados entre sí. Un ejemplo de esto puede ser crear todos los objetos relacionados con la configuración de un ambiente de pruebas, contrastado con los objetos para configurar un ambiente de desarrollo o producción, que suelen ser diferentes.

Definición: El patrón Abstract Factory provee una interfaz para crear familias de objetos relacionados o dependientes sin especificar su clase concreta.

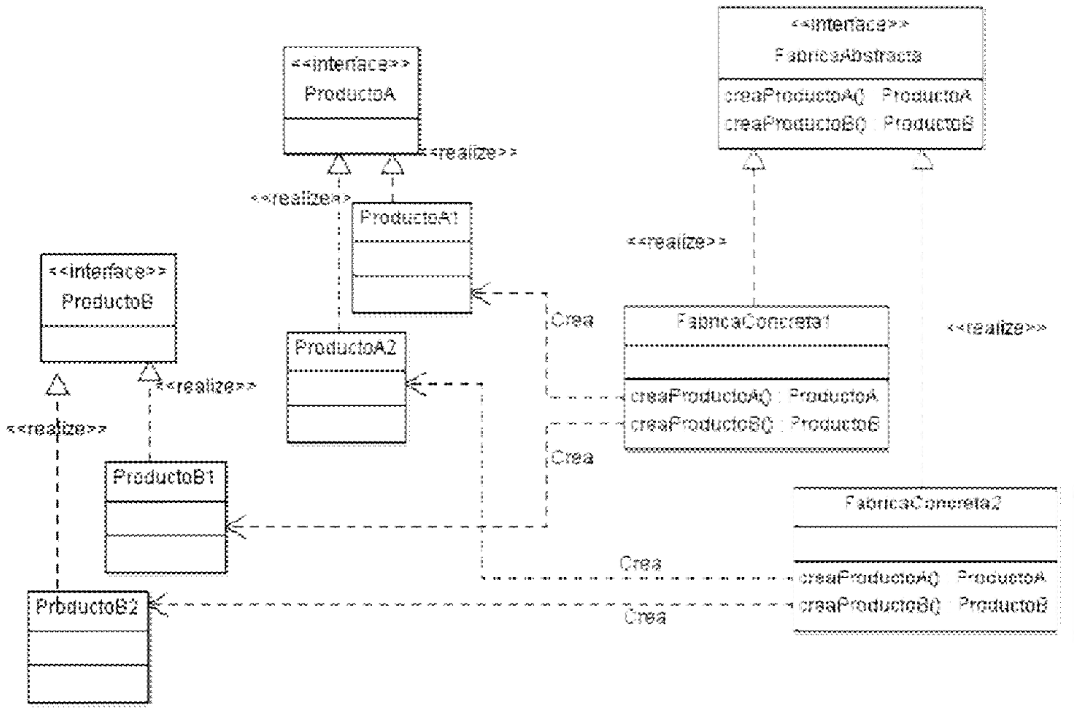

FIGURA IV.4 - DIAGRAMA DE CLASES: ABSTRACT FACTORY

`ProductoA` y `ProductoB` definen los tipos de productos generados por la *`FabricaAbstracta`*.

La `FabricaAbstracta` define un conjunto de métodos para producir productos, en donde cada uno crea un tipo de producto (A y B).

`ProductoA1` y `ProductoA2` son los posibles productos concretos del método *`creaProductoA()`*.

`ProductoB1` y `ProductoB2` son los posibles productos concretos del método *`creaProductoB()`*.

Las `FabricasConcretas` (1 y 2) implementan a la `FabricaAbstracta` para producir una familia de productos relacionados.

En este ejemplo, la `FabricaConcreta1` producirá la familia de objetos tipo 1 (`ProductoA1` y `ProductoB1`), mientras que la `FabricaConcreta2` producirá la familia de objetos tipo 2 (`ProductoA2` y `ProductoB2`).

El código cliente se escribe contra la `FabricaAbstracta` y los Productos abstractos, por lo que queda completamente desacoplado de los productos concretos, pudiendo configurar (e incluso cambiar) la `FabricaConcreta` a usar, en tiempo de ejecución.

```
// Configurado en algún lado
FabricaAbstracta fabrica = new FabricaConcreta2();

// Crea productos tipo 2 (ProductoA2 y ProductoB2)
ProductoA productoA = fabrica.creaProductoA();
ProductoB productoB = fabrica.creaProductoB();
```

```
fabrica = new FabricaConcreta1();
// Crea productos tipo 1 (ProductoA1 y ProductoB1)
productoA = fabrica.creaProductoA();
productoB = fabrica.creaProductoB();
```

OBJECT POOL

Problema: Algunos objetos son costosos de crear (en tiempo de ejecución, ciclos de procesador y/o memoria ocupada), por lo que se debe procurar crear pocos de éstos y tenerlos disponibles para cuando sean necesarios.

Definición: Este patrón permite reusar y compartir objetos que son costosos de crear.

Es sobretodo efectivo cuando el costo de inicializar la instancia de una clase es alto, la tasa de creación de instancias es alta y el número de instancias usadas concurrentemente es bajo.

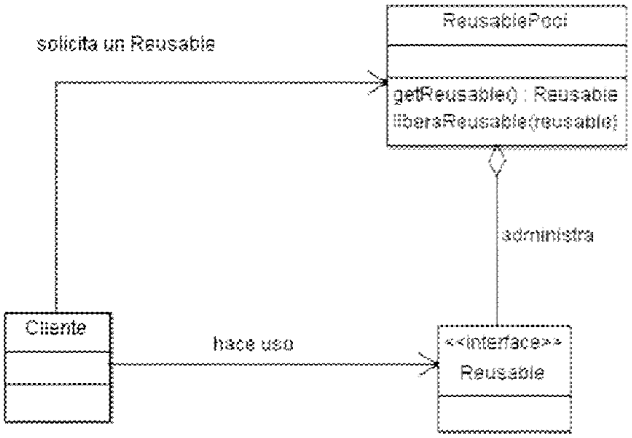

FIGURA IV.5 - DIAGRAMA DE CLASES: OBJECT POOL

Reusable: Interfaz que define el tipo de objetos manejados por el pool.

ReusablePool: Objeto (singleton) que administra al grupo de objetos reusables.

Cuenta con el método *`getReusable()`* que devuelve el primer objeto *`Reusable`* que tenga disponible, si no lo tiene puede crearlo y si no puede crearlo puede esperar a que se libere uno o lanzar una `Exception`. El objeto que devuelve lo quita del pool para que no esté disponible en la siguiente invocación.

También cuenta con el método *`liberaReusable()`*, que devuelve al pool un objeto asignado previamente para ponerlo como disponible nuevamente.

Cliente: Hace uso del Pool para obtener una instancia del *`ObjetoReusable`*. Es su responsabilidad regresar al pool el objeto que tomó, en cuanto lo haya terminado de usar, para hacerlo disponible.

Una implementación de este patrón podría verse de la siguiente forma:

```
public class Reusable {
    private String datosInternos = "";

    public Reusable() {
        logger.info("Creando Reusable");
        // Proceso de creación/inicialización
        // Toma mucho tiempo
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        logger.info("Reusable Creado");
    }

    public String getDatosInternos() {
        return datosInternos;
    }

    public void setDatosInternos(String datosInternos) {
        this.datosInternos = datosInternos;
    }
}

public class ReusablePool {
    private static int TAMANO_POOL = 3;

    private static List<Reusable> disponibles = new
ArrayList<Reusable>();
    private static List<Reusable> enUso = new ArrayList<Reusable>();

    public ReusablePool() {
        disponibles = new ArrayList<Reusable>(TAMANO_POOL);
        enUso = new ArrayList<Reusable>(TAMANO_POOL);

        for(int i=0; i<TAMANO_POOL; i++) {
            disponibles.add(new Reusable());
        }
    }

    public Reusable getReusable() {
        Reusable elegido;
        if(disponibles.size() > 0) {
            elegido = disponibles.remove(0);
        } else {
            elegido = new Reusable();
        }

        enUso.add(elegido);
        return elegido;
    }

    public void liberaReusable(Reusable reusable) {
        enUso.remove(reusable);
        limpiaReusable(reusable);

        if(disponibles.size() < TAMANO_POOL) {
```

```
            disponibles.add(reusable);
        }
    }

    private void limpiaReusable(Reusable reusable) {
        reusable.setDatosInternos("");
    }


}

public class ReusablePoolTest {
    private static final int USOS = 5;
    private static ReusablePool pool;

    public static void main(String[] args) {
        logger.info("Inicializando");
        pool = new ReusablePool();
        logger.info("Fin de Inicializacion");

        logger.info("Haciendo Uso");
        for(int i = 0; i<USOS; i++) {
            usaReusable();
        }
        logger.info("Fin del Uso");
    }

    private static void usaReusable() {
        logger.info("Obteniendo Reutilizable");
        Reusable reusable = pool.getReusable();
        reusable.setDatosInternos("info importante");
        String paraGuardar = reusable.getDatosInternos();
        pool.liberaReusable(reusable);
        logger.info("Fin del Reutilizable");
    }
}
```

Al ejecutar este código, tenemos la siguiente salida:

```
15:27:27:6 - Inicializando
15:27:27:7 - Creando Reusable
15:27:27:507 - Reusable Creado
15:27:27:507 - Creando Reusable
15:27:28:7 - Reusable Creado
15:27:28:7 - Creando Reusable
15:27:28:507 - Reusable Creado
15:27:28:507 - Fin de Inicializacion
15:27:28:507 - Haciendo Uso
15:27:28:507 - Obteniendo Reutilizable
15:27:28:507 - Fin del Reutilizable
15:27:28:507 - Obteniendo Reutilizable
15:27:28:507 - Fin del Reutilizable
15:27:28:507 - Obteniendo Reutilizable
15:27:28:507 - Fin del Reutilizable
15:27:28:507 - Obteniendo Reutilizable
15:27:28:507 - Fin del Reutilizable
15:27:28:507 - Obteniendo Reutilizable
```

```
15:27:28:507 - Fin del Reutilizable
15:27:28:507 - Fin del Uso
```

En donde podemos apreciar que el proceso de inicialización es tardado, ya que le toma 500 milisegundos crear cada uno de los objetos Reusable almacenados en el pool.

Sin embargo, después de creados, el obtener instancias de estos objetos y realizar operaciones sobre ellos es prácticamente inmediato y esa es la ventaja principal de este patrón, que los objetos estén disponibles para cuando se requieran y no se pierda tiempo y recursos valiosos creándolos.

COMPOSITE

Problema: Existe en la naturaleza (y en el software) información que se puede modelar de mejor manera con una representación en forma de árbol. Ejemplos de esto es la clasificación que existe del reino animal, o que puedan existir menús con submenús de varios niveles.

Para obtener información de estos “árboles” es necesario recorrerlo, visitando cada nodo, uno por uno. Si existen distintos tipos de nodos, deberemos entonces escribir código para diferenciarlos y ejecutar lógica distinta para cada tipo, lo que añade complejidad al código cliente que hace uso de la estructura.

Definición: El patrón Composite permite agrupar objetos en estructuras de árbol para representar jerarquías (parciales o totales), permitiendo tratar objetos individuales y composiciones de forma uniforme.

En otras palabras, en la mayoría de los casos, es posible “ignorar” las diferencias que puedan existir entre los distintos tipos de elementos que conforman la estructura, lo que simplifica trabajar con estos objetos.

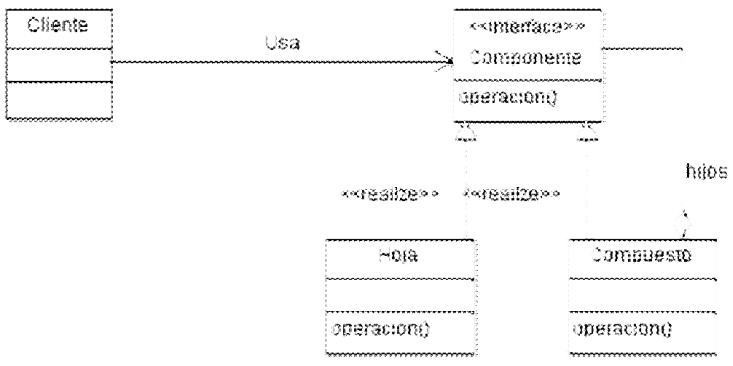

FIGURA IV.6 - DIAGRAMA DE CLASES: PATRÓN COMPOSITE

Componente: Define una interfaz común a todos los objetos en la estructura, tanto nodos hoja como nodos compuestos.

Hoja: Define el comportamiento de los elementos individuales al implementar las operaciones definidas por Componente.

Compuesto: Define el comportamiento de los componentes compuestos de otros componentes implementando las operaciones definidas por Componente. A los nodos que lo componen, se les suele denominar “hijos”, en la estructura jerárquica.

Cliente: Código cliente que hace uso de la estructura a través de la interfaz Componente, con lo que se oculta si el nodo en cuestión es una Hoja o un Compuesto.

Una implementación de este patrón en Java se podría ver de la siguiente forma:

```
public interface Componente {
    void operacion();
}

public class Hoja implements Componente {
    private String datosInternos;

    public Hoja(String datosInternos) {
        this.datosInternos = datosInternos;
    }

    @Override
    public void operacion() {
        System.out.println("Hoja: " + datosInternos);
    }
}

public class Compuesto implements Componente {
    private List<Componente> hijos;

    public Compuesto(Componente ... hijos) {
        this.hijos = Arrays.asList(hijos);
    }

    @Override
    public void operacion() {
        System.out.println("Compuesto de " + hijos.size() + " hijos");
        System.out.println("----------------------------");
        for (Componente componente : hijos) {
            System.out.print("\t");
            componente.operacion();
        }
    }
}
```

El código cliente puede crear un elemento individual:

```
        Componente hoja1 = new Hoja("1");
        hoja1.operacion();
```

Con lo que la salida sería:

```
      Hoja: 1
```

Pero de forma análoga, podría ahora crear y usar un elemento compuesto de dos elementos individuales:

```
Componente hoja2 = new Hoja("2");

Compuesto compuesto = new Compuesto(hoja1, hoja2);
compuesto.operacion();
```

Dando la siguiente salida:

```
Compuesto de 2 hijos
---------------------------
      Hoja: 1
      Hoja: 2
```

Incluso el patrón permite crear otro compuesto, compuesto a su vez de 2 elementos simples y el compuesto creado anteriormente.

```
Compuesto otroCompuesto = new Compuesto(hoja1, hoja2, compuesto);
otroCompuesto.operacion();
```

Y esta sería la salida del código:

```
Compuesto de 3 hijos
---------------------------
      Hoja: 1
      Hoja: 2
      Compuesto de 2 hijos
---------------------------
      Hoja: 1
      Hoja: 2
```

TEMPLATE METHOD

Problema: El copiar y pegar extractos de código en dos (o más) componentes que presentan un comportamiento muy similar, pero con pequeñas diferencias bien identificadas, es una práctica común en el desarrollo de software. Sin embargo esta práctica puede introducir problemas de mantenimiento a largo plazo, ya que duplica la cantidad de código, y entonces al requerir un cambio o detectar un error, se debe cambiar en todas las copias.

Definición: El patrón Template Method, define los pasos de un algoritmo y permite a las subclases proveer la implementación de uno o más de los pasos, a la vez que no permite a las subclases cambiar la estructura básica del algoritmo.

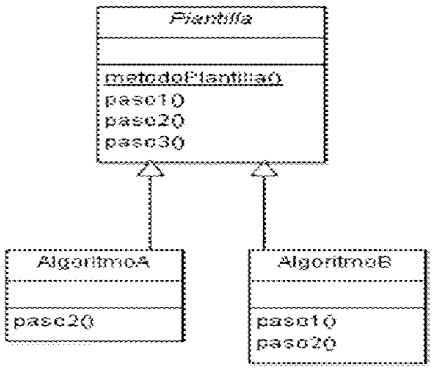

FIGURA IV.7 - DIAGRAMA DE CLASE: TEMPLATE METHOD

`Plantilla` es la clase abstracta que define un `metodoPlantilla()`, que define los pasos de un algoritmo, en donde algunos de esos pasos, son métodos abstractos a ser implementados por las subclases. Este método suele ser final, para no poder ser cambiado por ninguna de las subclases.

`AlgoritmoA` y `AlgorimoB` son dos clases concretas, en donde cada una realiza una implementación completa del algoritmo definido por la `Plantilla`.

Con el fin de tener un mejor control sobre los pasos del algoritmo, la clase Plantilla puede definir 3 tipos de métodos:

- Método abstracto: Es un método sin implementación, de modo que la subclase queda obligado a proveer una implementación propia.
- Método concreto: Es un método concreto y final, de tal forma que la subclase no pueda sobrescribirlo.
- Método gancho: Es un método concreto, normalmente vacío o con una implementación default, pero puede ser sobrescrito por la subclase.

Una posible implementación de este patrón sería la siguiente:

```
public abstract class Plantilla {
    // Método Plantilla: Algorimo básico
    public final void metodoPlantilla() {
        paso1();
        paso2();
        paso3();
    }

    // Método Gancho: Tiene una Implementación por defecto
    public void paso1() {
        System.out.print("> ");
    }

    // Método Abstracto: Es obligatorio implementarlo
    public abstract void paso2();

    // Método Concreto: Imposible cambiar este paso
```

```
    public final void paso3() {
        System.out.print(".\n");
    }
}
```

En la clase `Plantilla` se está definiendo un algoritmo que consta de invocar secuencialmente tres métodos (por supuesto, este algoritmo podría ser mucho más complicado si fuera necesario).

Aquí cada uno de los pasos demuestra uno de los tipos de métodos que la plantilla puede definir.

Finalmente se definen un par de implementaciones concretas de la `Plantilla`.

```
public class AlgoritmoA extends Plantilla {

    public void paso2() {
        System.out.print("Saludos desde el Algoritmo A");
    }

}

public class AlgoritmoB extends Plantilla {

    public void paso1() {
        System.out.print("# ");
    }

    public void paso2() {
        System.out.print("Esto llega del Algoritmo B");
    }

}
```

Al ejecutar este código

```
        Plantilla algoritmoA = new AlgoritmoA();
        algoritmoA.metodoPlantilla();

        Plantilla algoritmoB = new AlgoritmoB();
        algoritmoB.metodoPlantilla();
```

Se muestra la salida del programa a continuación:

```
> Saludos desde el Algoritmo A.
# Esto llega del Algoritmo B.
```

Se puede observar que los dos algoritmos (A y B) terminan en un punto, puesto que el paso 3 no se puede cambiar, que el `AlgoritmoA` empieza con un carácter '>', puesto que es el comportamiento por defecto de la plantilla, mientras que el `AlgoritmoB` cambio ese comportamiento para iniciar con un carácter '#' y finalmente ambos algoritmos fueron obligados a definir el paso 2, en el que imprimieron un mensaje.

Este patrón aplica el principio Hollywood (No nos llames, nosotros te llamamos), ya que la clase padre define el algoritmo general y decide cuando ejecutar algún método abstracto, concreto o de gancho del hijo.

PROXY

Problema: Un software siempre es propenso al cambio, ya sea para corregir errores introducidos en el o por cambios en la operación del negocio, pero ¿cómo hacer para introducir nueva funcionalidad sin tener que tocar la funcionalidad existente?

Por ejemplo, si en un sistema ya se realizan operaciones de todo tipo con los usuarios, y ahora se quiere agregar seguridad que evite que esas operaciones sean realizadas sin los permisos necesarios. Se podría modificar cada operación y revisar las credenciales del usuario, pero sería más conveniente poder hacerlo sin ser tan intrusivo y potencialmente afectar las operaciones en sí.

Definición: Un proxy provee un objeto como sustituto de otro para controlar el acceso a este último. Puede ser debido a que el objeto a acceder es remoto, costoso de crear o requiere de seguridad.

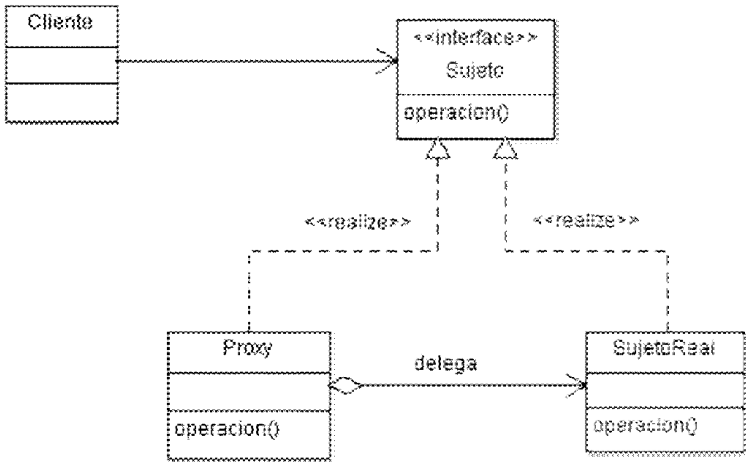

FIGURA IV.8 - DIAGRAMA DE CLASES: PROXY

La interfaz `Sujeto` provee una interfaz que tanto el `SujetoReal` como el `Proxy` deberán implementar. De esta manera el `Proxy` puede sustituir al `SujetoReal` en cualquier situación.

El `SujetoReal` es el que realiza el trabajo real para el que fue diseñado en la aplicación (contiene lógica de negocio), y no debe tener conocimiento del `Proxy`.

El `Proxy` representa al `SujetoReal`, recibe las peticiones del `Cliente` y las reenvía al `SujetoReal`, pudiendo agregar funcionalidad antes y/o después de invocar la operación.

Para ello, el `Proxy` mantiene una referencia al `SujetoReal`, controlando el acceso a éste y puede ser responsable de su creación y destrucción en el momento adecuado.

El cliente mantendrá una referencia a la interfaz `Sujeto`, por lo que no sabrá si el objeto es un proxy o es real.

Una implementación del diagrama anterior podría ser la siguiente:

```
public interface Sujeto {
    void operacion();
}

public class SujetoReal implements Sujeto {
    public SujetoReal() {
        System.out.println("Creando Sujeto Real");
    }

    @Override
    public void operacion() {
        System.out.println("Realizando Operación Real");
    }
}


public class Proxy implements Sujeto {
    private Sujeto delegado;

    public Proxy() {
        System.out.println("Creando Proxy");
    }

    @Override
    public void operacion() {
        System.out.println("Proxy antes de operacion()");
        getDelegado().operacion();
        System.out.println("Proxy después de operacion()");
    }

    private Sujeto getDelegado() {
        if(delegado == null) {
            delegado = new SujetoReal();
        }

        return delegado;
    }
}
```

En el código cliente se puede codificar lo siguiente:

```
        Sujeto sujeto = new Proxy();

        sujeto.operacion();
```

Con lo que la salida de ejecutar este programa es:

```
Creando Proxy
Proxy antes de operacion()
Creando Sujeto Real
Realizando Operación Real
Proxy después de operacion()
```

Lo que destaca de un proxy es su capacidad de agregar funcionalidad nueva a código existente, de forma prácticamente transparente. Sin embargo, ésta funcionalidad suele no estar relacionada con la funcionalidad del objeto real, sino que es complementaria y factible de mantener separada, lo que a su vez, ayuda al mantenimiento.

Un proxy puede ser clasificado por la funcionalidad que proporciona en:

- Remote Proxy: Controla el acceso a un objeto remoto.
- Virtual Proxy: Controla el acceso a un objeto costoso de crear.
- Protection Proxy: Controla el acceso a un recurso basado en privilegios.
- Firewall Proxy: Controla el acceso a un grupo de recursos de red, protegiéndolos de "malos" clientes.
- Smart Reference Proxy: Provee funcionalidad adicional al referenciar objetos. Por ejemplo, contar el número de referencias que un objeto tiene.
- Caching Proxy: Provee un almacenamiento temporal de resultados costosos. También permite a varios clientes compartir los resultados, reduciendo así el tiempo de procesamiento o el ancho de banda.
- Synchronization Proxy: Provee un acceso seguro a un recurso, cuando se realiza desde varios hilos, evitando conflictos por usar el recurso de forma simultánea.
- Complexity Hiding Proxy: Esconde la complejidad y los controles de acceso a un grupo complejo de clases.

OBSERVER

Problema: Para que un sistema funcione correctamente, debe haber una comunicación eficiente entre sus distintos componentes. Muchas veces existen componentes que están interesados en enterarse de que el estado de otro componente ha cambiado.

Una forma tradicional, pero poco eficiente, de hacer esto es que los componentes interesados estén preguntando cada cierto tiempo al sujeto de interés si su estado ha cambiado (polling).

Otra forma es que el sujeto de interés avise a sus dependientes que su estado ha cambiado hasta el momento en que eso ocurre, esto es más eficiente, pero sin una interfaz común para avisar a todos sus dependientes se dificulta el mantenimiento al ir agregando más dependientes distintos a lo largo del tiempo.

Definición: El patrón Observer define una relación de dependencia de uno-a-muchos tal que, cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

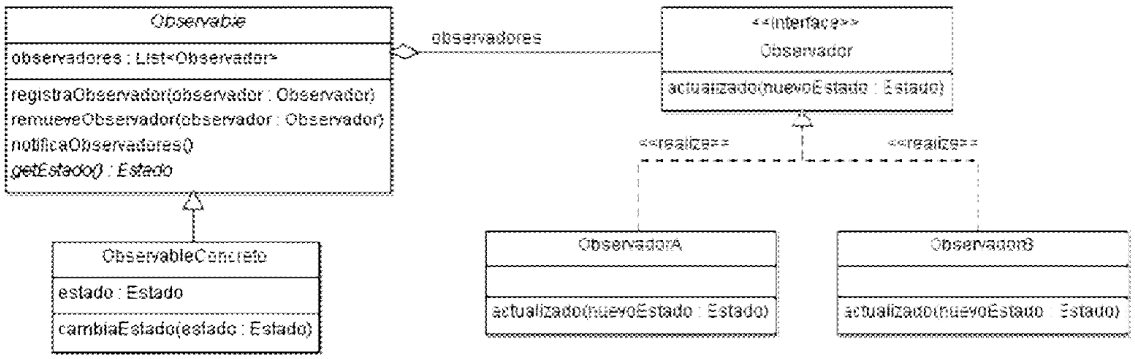

FIGURA IV.9 - DIAGRAMA DE CLASES: PATRÓN OBSERVER

`Observador`: Define una interfaz para recibir notificaciones de actualización.

`Observable`: Representa al sujeto de interés. Implementa la funcionalidad de registrar, remover y notificar a los interesados (Observadores).

`ObservableConcreto`: Es el sujeto real de interés. Mantiene un estado de datos que al cambiar, ejecuta la rutina que notifica a todos los observadores registrados.

`ObservadorA` y `ObservadorB`: Son los observadores concretos interesados en recibir los cambios en el estado de ObservableConcreto. Implementan la interfaz Observador en cuyos métodos definen la acción a realizar al recibir una actualización.

Para que un observador pueda empezar a recibir notificaciones de parte del `Observable`, basta con que se registre a él usando el método `registraObservador()`, después de eso, cualquier invocación al método `cambiaEstado()` del `ObservableConcreto`, ejecutará el método `notificaObservadores()`, que recorre la lista de observadores registrados, invocando el método `actualizado()` sobre cada uno de ellos, lo que se traduce en que cada uno de los observador fue notificado del cambio.

El patrón Observer, promueve el bajo acoplamiento entre los distintos componentes de un sistema, puesto que permite que el sujeto `Observable` envíe mensajes a todos sus dependientes con sólo saber que son de tipo `Observador`, lo que elimina que tenga que notificar de forma diferente a diferentes interesados.

Una implementación del patrón Observer podría verse de la siguiente manera, empezando con definir la interfaz para todos los observadores interesados:

```
public interface Observador {

  public void actualizado(Estado nuevoEstado);

}
```

Después se implementa la funcionalidad común que permite a registrar y notificar a todos los observadores de un cambio:

```
public abstract class Observable {

    public List<Observador> observadores = new ArrayList<Observador>();

    public void registraObservador(Observador observador) {
        observadores.add(observador);
    }

    public void remueveObservador(Observador observador) {
        observadores.remove(observador);
    }

    public void notificaObservadores() {
        for (Observador observador : observadores) {
            observador.actualizado(getEstado());
        }

    }

    public abstract Estado getEstado();

}

public class ObservableConcreto extends Observable {

    private Estado estado;

    public void cambiaEstado(Estado estado) {
        this.estado = estado;

        notificaObservadores();
    }

    @Override
    public Estado getEstado() {
        return estado;
    }
}
```

Implementaciones concretas de observadores interesados:

```
public class ObservadorA implements Observador {

    public void actualizado(Estado nuevoEstado) {
        System.out.println("ObservadorA enterado del cambio: " +
                                        nuevoEstado.getMensaje());
    }

}

public class ObservadorB implements Observador {

  public void actualizado(Estado nuevoEstado) {
      System.out.println("Como interesado del Sujeto, ObservadorB es
enterado del cambio");
      System.out.print("Y hace algo con la información nueva: " +
                                        nuevoEstado.getMensaje());
  }
```

```
}
```

En el código cliente se podría ejecutar el siguiente código:

```
        ObservableConcreto sujeto = new ObservableConcreto();
        sujeto.cambiaEstado(new Estado("Estado Inicial"));

        Observador observadorA = new ObservadorA();
        sujeto.registraObservador(observadorA);

        Observador observadorB = new ObservadorB();
        sujeto.registraObservador(observadorB);

        sujeto.cambiaEstado(new Estado("Nuevo Estado"));
```

Y la salida del programa imprime lo siguiente al recibir la notificación de cambio en cada uno de los observadores:

```
ObservadorA enterado del cambio: Nuevo Estado

Como interesado del Sujeto, ObservadorB es enterado del cambio
Y hace algo con la información nueva: Nuevo Estado
```

STRATEGY

Problema: Durante el desarrollo de un proyecto se codifican algoritmos que realizan las acciones apropiadas a una situación. Es común que la acción a ejecutar sea diferente dependiendo de algunas variables en el contexto, lo que se suele traducir en un código monolítico con una combinación de condicionales que aglutinan distintos algoritmos para realizar una acción. Este tipo de código puede resultar en ser difícil de leer, lo que complica su mantenimiento, sobre todo si con el tiempo se agregan más condicionales y algoritmos.

Definición: El patrón Strategy define una familia de algoritmos, encapsula cada uno, y los hace intercambiables.

De esta forma, el patrón pretende separar los algoritmos en clases distintas tal que cada uno pueda variar sin afectar a los demás, al tiempo que permite elegir en tiempo de ejecución el algoritmo apropiado a ejecutar.

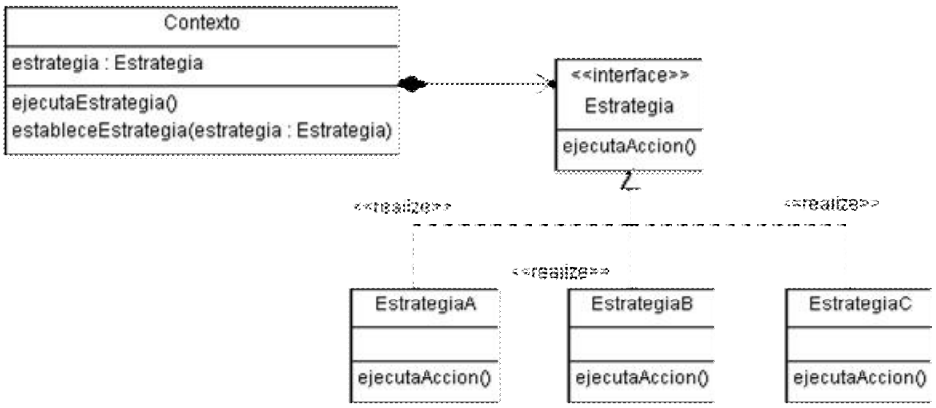

FIGURA IV.10 - STRATEGY PATTERN

`Estrategia`: Representa a toda la familia de algoritmos, definiendo una interfaz que todas las estrategias concretas implementarán de forma independiente (respecto a otras).

`Contexto`: Es quien hace uso de la Estrategia, por lo que mantiene una referencia a ella, pudiendo usar cualquiera de las estrategias concretas de la familia.

`EstrategiaA`, `EstrategiaB` y `EstrategiaC`: Son los algoritmos concretos a utilizar. Se enfocan en resolver una problemática. En ocasiones, los algoritmos requerirán datos con los cuales operar, estos pueden llegar en forma de parámetros en el método `ejecutaAccion()` o incluso pueden tener acceso al `Contexto` y solicitar a este los datos.

En seguida se muestra una implementación del Patrón Strategy.

```
public class Contexto {
    public Estrategia estrategia = new EstrategiaC();

    public void ejecutaEstrategia() {
        estrategia.ejecutaAccion();
```

```
    }

    public void estableceEstrategia(Estrategia estrategia) {
        this.estrategia = estrategia;
    }
}

public interface Estrategia {
    public void ejecutaAccion();
}

public class EstrategiaA implements Estrategia {
    public void ejecutaAccion() {
        System.out.println("Ejecutando Estrategia A: Defensiva");
    }
}
public class EstrategiaB implements Estrategia {
    public void ejecutaAccion() {
        System.out.println("Ejecutando Estrategia B: Ofensiva");
    }
}
public class EstrategiaC implements Estrategia {
    public void ejecutaAccion() {
        System.out.println("Ejecutando Estrategia C: Normal");
    }
}
```

El código anterior se podría ejecutar de la siguiente forma en un cliente:

```
        Contexto contexto = new Contexto();
        contexto.ejecutaEstrategia();

        contexto.estableceEstrategia(new EstrategiaA());
        contexto.ejecutaEstrategia();

        contexto.estableceEstrategia(new EstrategiaB());
        contexto.ejecutaEstrategia();
```

El resultado es el siguiente texto, resultado de usar sobre el mismo contexto los tres algoritmos definidos (A, B y C):

```
Ejecutando Estrategia C: Normal
Ejecutando Estrategia A: Defensiva
Ejecutando Estrategia B: Ofensiva
```

MODEL VIEW CONTROLLER

Problema: Una aplicación compleja o de gran tamaño, puede ser difícil de mantener si el código que la compone tiene una estructura poco clara y mezcla lógica del negocio con lógica de pantalla y rutinas para comunicación y guardado de datos.

Definición: El patrón Modelo Vista Controlador separa las rutinas de una aplicación en estás tres capas con el fin de facilitar el desarrollo, pero sobretodo el mantenimiento.

Es en realidad un patrón **compuesto** de otros patrones: Observer, Strategy y Composite. Estos patrones trabajan juntos para desacoplar los 3 componentes (modelo, vista y controlador), lo que mantiene el diseño claro y flexible.

- Modelo: Contiene toda la lógica para acceder y cambiar los datos. Hace uso del patrón Observer para informar a los interesados (la vista) que algún dato ha cambiado, por lo que se mantiene totalmente desacoplado del resto de la aplicación.

- Vista: Se encarga de crear y actualizar la interfaz de usuario. También es el encargado de recibir las interacciones del usuario en la interfaz

    - Crear: Para crear la interfaz de usuario es común usar el patrón Composite, que permite anidar componentes (como campos de texto y botones) dentro de otros (contenedores).

    - Actualizar: Para actualizar la interfaz de usuario, normalmente se hace uso del patrón Observer, haciendo que la vista se registre como observador para ser notificado en cuanto haya cambios en el modelo.

- Controlador: Es el mediador entre la vista y el modelo. Recibe las acciones realizadas por el usuario en la interfaz de usuario, las procesa y decide qué acciones realizar sobre el modelo.

Usa el patrón Strategy para desacoplarse de la vista. Con este patrón, la vista accede a la funcionalidad del controlador mediante una interfaz, que puede tener diferentes implementaciones para así lograr tener comportamientos diferentes.

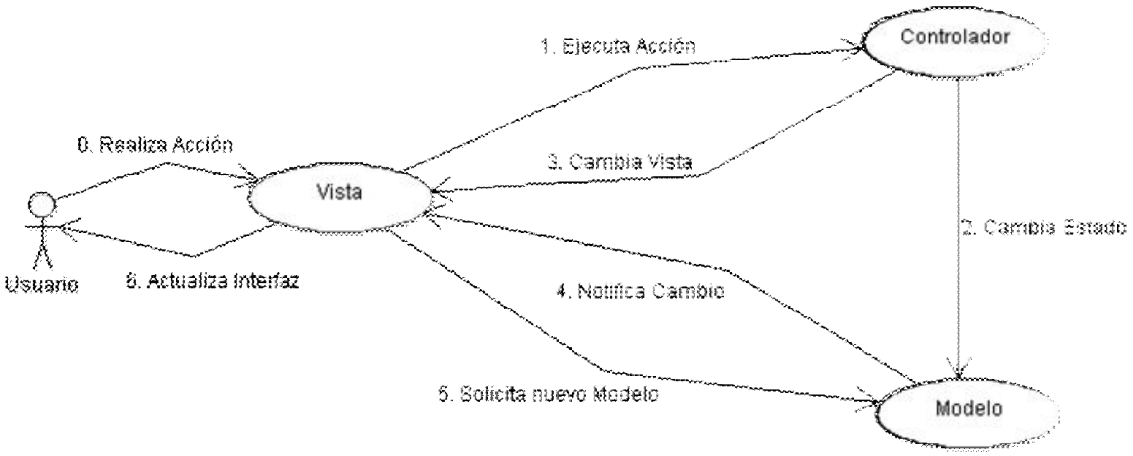

FIGURA IV.11 - INTERACCIONES ENTRE EL MODELO, LA VISTA Y EL CONTROLADOR

0. Realiza acción: El usuario realiza una acción en la interfaz gráfica (vista), teclea información o da click en un botón.

1. Ejecuta acción: La vista informa al controlador lo que el usuario hizo, y éste traduce el gesto en una acción concreta a ejecutar.

2. Cambia estado: El controlador solicita al modelo ejecutar la lógica de negocio que tiene como resultado cambiar su estado para cumplir con la acción que el usuario ha solicitado. Esta acción puede ser crear un nuevo registro, guardar cambios en el estado, etc.

3. Cambia vista: El controlador podría solicitar un cambio a la vista como consecuencia de realizar la acción solicitada, como podría ser habilitar o deshabilitar botones u opciones de menú.

4. Notifica cambio: Al terminar la ejecución, el modelo notifica a la vista que su estado interno ha cambiado (ya sea como resultado de la acción solicitada o algún otro cambio interno).

5. Solicita nuevo modelo: La vista solicita del modelo los nuevos datos que debe desplegar.

6. Actualiza interfaz: La interfaz gráfica se actualiza para mostrar los nuevos datos, por lo que el usuario observa en la pantalla los resultados que espera.

Es importante resaltar que a pesar del nombre “modelo”, en este componente no sólo hay datos, sino que incluye la lógica real de la aplicación para manipular esos datos y es común exponer esto a través de servicios que proveen un API bien definida.

El controlador sólo implementa la lógica de la vista, respondiendo a acciones del usuario y manda llamar la lógica apropiada en el modelo (que es quien realmente debe realizar los cambios en los datos de acuerdo a la lógica del negocio).

Esto permite que aunque se decida cambiar la interfaz gráfica del sistema, se pueda reusar toda la lógica del negocio contenida en el modelo. Incluso permite que existan varias interfaces gráficas que interactúen al mismo tiempo (por ejemplo en un sistema que se debe poder acceder desde una computadora personal y un dispositivo móvil).

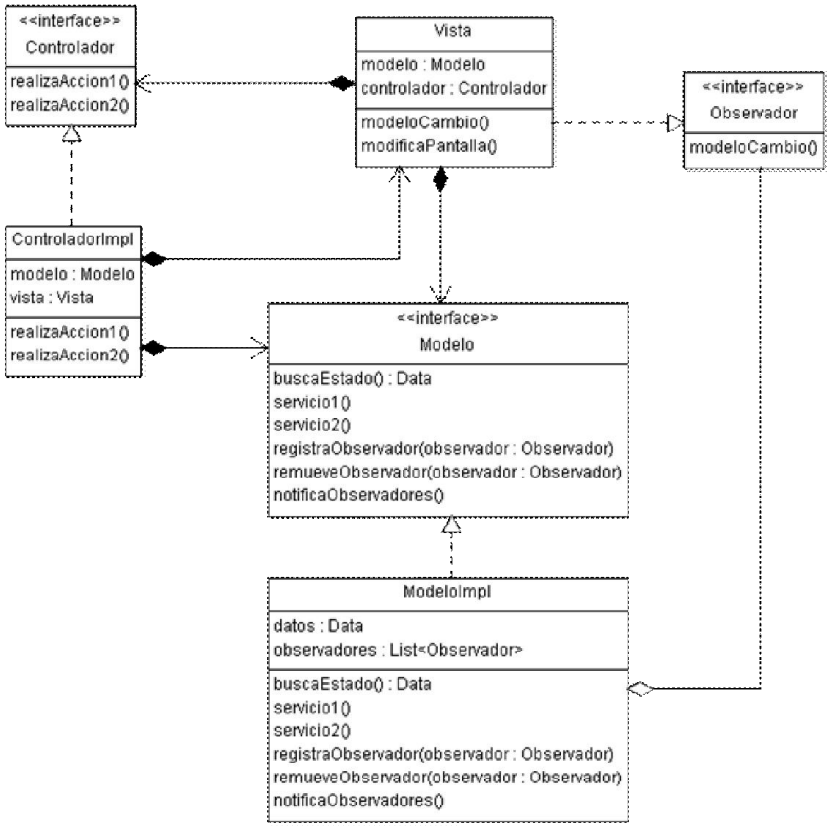

FIGURA IV.12 - DIAGRAMA DE CLASES: MODELO VISTA CONTROLADOR

A continuación se lista una posible implementación del diagrama MVC mostrado. Empezando con el modelo, ya que es el componente con menos dependencias:

```
public interface Modelo {

    public void registraObservador(Observador observador);

    public void remueveObservador(Observador observador);

    public Data buscaEstado();

    public void servicio1();

    public void servicio2();

    public class Data {
        private Integer contador;

        public Data() {
            contador = 0;
        }

        public Integer getContador() {
            return contador;
        }

        public void setContador(Integer contador) {
            this.contador = contador;
```

```
        }
    }
}
```

Esta interfaz del `Modelo`, contiene por una parte métodos para registrar observadores (a los que posteriormente les puede notificar de los cambios ocurridos) y por otra los métodos que realizan acciones que modifican los datos del modelo (`servicio1` y `servicio2`). Adicionalmente se define el método `buscaEstado()`, a través del cual la pantalla puede solicitar el estado actual del modelo para mostrarlos gráficamente en la vista.

`Data` representa el estado actual del modelo de datos, ejemplificado aquí con un contador.

```
public class ModeloImpl implements Modelo {

    public Data datos;

    public List<Observador> observadores;

    public ModeloImpl() {
        datos = new Data();
        observadores = new ArrayList<Observador>();
    }

    public void registraObservador(Observador observador) {
        observadores.add(observador);
    }

    public void remueveObservador(Observador observador) {
        observadores.remove(observador);
    }

    private void notificaObservadores() {
        System.out.println("Modelo: El estado ha cambiado, notificando
interesados...");
        for (Observador observador : observadores) {
            observador.modeloCambio();
        }
    }

    public Data buscaEstado() {
        return datos;
    }

    public void servicio1() {
        System.out.println("Modelo: Ejecutando Logica de Negocio para
incrementar contador");
        datos.setContador(datos.getContador() + 1);

        notificaObservadores(); // Al final notifica cambios
    }

    public void servicio2() {
        System.out.println("Modelo: Ejecutando Logica de Negocio para
disminuir contador");
        datos.setContador(datos.getContador() - 1);
```

```
        notificaObservadores(); // Al final notifica cambios
    }

}
```

La interfaz `Observador` define el método `modeloCambio()` que será invocado cuando el `Modelo` detecte un cambio que deba ser notificado a los interesados registrados.

```
public interface Observador {

    public void modeloCambio();

}
```

A continuación se implementa la `Vista`, que representa la parte de la aplicación que interactúa directamente con el usuario. Por simplicidad no se incluyó una interfaz gráfica de usuario, pero bien podría haberse usado Swing o JavaFX para este fin.

```
public class Vista implements Observador {

    public Modelo modelo;

    public Controlador controlador;

    private int contador;

    public Vista(Modelo modelo, Controlador controlador) {
        this.modelo = modelo;
        this.controlador = controlador;

        modelo.registraObservador(this);
    }

    public void creaPantalla() {
        System.out.println("Vista: Construyendo Pantalla Inicial...");
        contador = modelo.buscaEstado().getContador();
        modificaPantalla();
    }

    public void modeloCambio() {
        System.out.println("Vista: El modelo ha cambiado, actualizando
pantalla...");
        contador = modelo.buscaEstado().getContador();
        modificaPantalla();
        System.out.println("Vista: Pantalla actualizada. Nuevo valor
desplegado: " + contador);
    }

    public void modificaPantalla() {
        System.out.println("Vista: Muestra Valor Actual: " + contador);
    }

    public void presionaBotonMas() {
        System.out.println("Vista: El usuario ha realizado una acción:
incrementa");
        controlador.realizaAccion1();
```

```
    }

    public void presionaBotonMenos() {
        System.out.println("Vista: El usuario ha realizado una acción:
incrementa");
        controlador.realizaAccion2();
    }
}
```

Finalmente se define el `Controlador`, que recibe acciones realizadas en la `Vista`, e implementa la acción a realizar para cada una de ellas.

```
public interface Controlador {

    public void realizaAccion1();

    public void realizaAccion2();

    Vista getVista();

}

public class ControladorImpl implements Controlador {

    public Modelo modelo;

    public Vista vista;

    public ControladorImpl(Modelo modelo) {
        this.modelo = modelo;
        vista = new Vista(modelo, this);
        vista.creaPantalla();
    }

    public Vista getVista() {
        return vista;
    }

    public void realizaAccion1() {
        System.out.println("Controlador: ejecutando la accion
incrementa");
        modelo.servicio1();
    }

    public void realizaAccion2() {
        System.out.println("Controlador: ejecutando la accion
decrementa");
        modelo.servicio2();
    }
}
```

En este caso, las acciones ejecutan servicios específicos del `Modelo`, que realizarán cambios en los datos.

Ahora es posible crear una pequeña prueba que nos ayudará a determinar si ocurre el comportamiento esperado.

```
public class MVCTest {
    public static void main(String[] args) {
        System.out.println("Inicializando sistema...");
        System.out.println("Construyendo Modelo de Datos...");
        Modelo modelo = new ModeloImpl();

        System.out.println("Construyendo Controlador de pantalla...");
        Controlador controlador = new ControladorImpl(modelo);

        System.out.println(">Sistema listo para operar");

        // Se simulan acciones de usuario sobre la vista
        Vista vista = controlador.getVista();
        vista.presionaBotonMas();
        vista.presionaBotonMenos();
    }
}
```

Salida de ejecución:

```
Inicializando sistema...
Construyendo Modelo de Datos...
Construyendo Controlador de pantalla...
Vista: Construyendo Pantalla Inicial...
Vista: Muestra Valor Actual: 0
>Sistema listo para operar
Vista: El usuario ha realizado una acción: incrementa
Controlador: ejecutando la accion incrementa
Modelo: Ejecutando Logica de Negocio para incrementar contador
Modelo: El estado ha cambiado, notificando interesados...
Vista: El modelo ha cambiado, actualizando pantalla...
Vista: Muestra Valor Actual: 1
Vista: Pantalla actualizada. Nuevo valor desplegado: 1
Vista: El usuario ha realizado una acción: incrementa
Controlador: ejecutando la accion decrementa
Modelo: Ejecutando Logica de Negocio para disminuir contador
Modelo: El estado ha cambiado, notificando interesados...
Vista: El modelo ha cambiado, actualizando pantalla...
Vista: Muestra Valor Actual: 0
Vista: Pantalla actualizada. Nuevo valor desplegado: 0
```

De la salida del programa se puede apreciar el orden en la creación de los componentes MVC y cómo al invocar acciones sobre la vista, se inicia el ciclo de notificaciones MVC para cambiar el estado del modelo y finalmente terminar en una actualización de la pantalla (vista).

MODEL 2

En el mundo de las aplicaciones web, después de agregar los templates Java Server Pages (JSP's) a la especificación de J2EE en 1998, parecía una buena idea tener pantallas dinámicas de html que generaran por sí mismas todo su contenido, ya que se tenía en un solo lugar todo lo relacionado a esa pantalla (componentes visuales, lógica de negocio, acceso a datos, etc.), lo que daba como resultado una arquitectura “modular”. A esta forma de programar se le llamó “Modelo 1”.

Mientras la aplicación desarrollada no tuviera pantallas muy complejas o el sistema no fuera muy grande, el “Modelo 1” funcionaba bastante bien. Sin embargo al crecer la complejidad, aumentaban el número de casos que requerían copiar y pegar código similar en pantallas diferentes y aumentaba también la cantidad de código en el JSP, dificultando la reusabilidad, el mantenimiento y la lectura del código.

Inspirada en la separación de responsabilidades del patrón MVC, pero modificado para adaptarse a la arquitectura de las aplicaciones web, el llamado “Modelo 2” (para distinguirse del Modelo 1), propone combinar las tecnologías de Servlet y JSP para lograr tener la misma separación de componentes que en una interfaz de usuario tradicional.

- El **modelo** de datos se encarga de proveer servicios para consultar y modificar los datos (lógica del negocio). No se especifica una tecnología en particular para este componente, pero se enfatiza realizar su separación del resto del código, puesto que es la parte más reutilizable del sistema, ya que distintas pantallas podrían compartir los mismos datos.

- La **vista** se encarga de la presentación de los datos simplificando el código y haciendo posible que un diseñador gráfico se encargue de esta capa. Se ocupan los Java Server Pages para desempeñar este trabajo, puesto que están basados en HTML que es un lenguaje para especificar pantallas web para ser desplegadas en un navegador.

- El **controlador** se ocupa de procesar las peticiones del usuario, consultar/ejecutar cambios en el modelo de datos y ejecutar la vista apropiada, pasándole la información que requiera desplegar. Se ocupan los Servlet para realizar esta función debido a que están diseñados para procesar programáticamente peticiones http, por lo que son ideales para recibir acciones de usuario, ejecutar código Java (en el modelo) y redireccionar hacia una vista para mostrar el resultado de la operación.

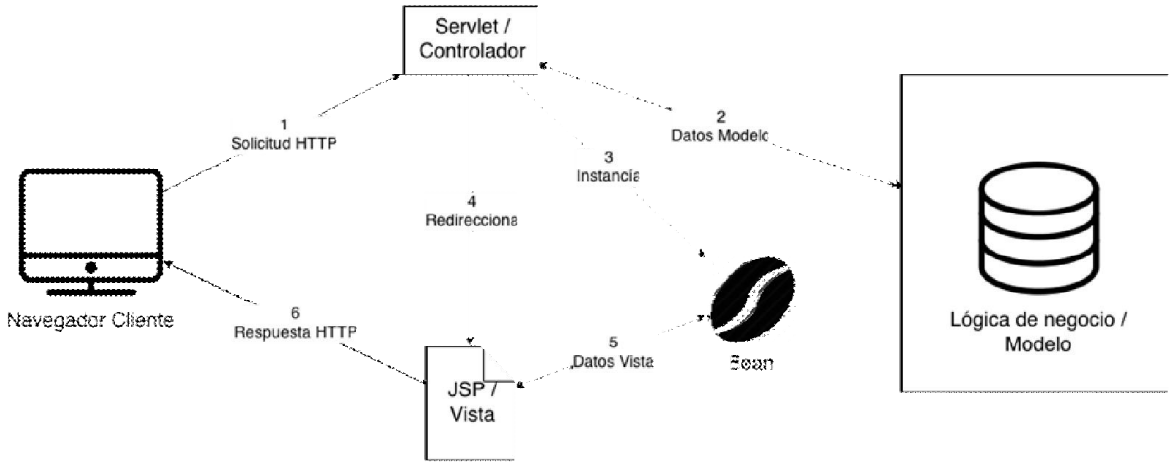

FIGURA IV.13 - MODELO 2

1. El cliente realiza una petición HTTP al servidor. La petición incluye acciones a realizar y los datos necesarios para ello.

2. El servidor recibe en un controlador (Servlet) la petición y la procesa, consultando datos del Modelo o ejecutando acciones que modifican su estado.

3. Los datos que se requieran mostrar en la pantalla se suelen guardar en un JavaBean.

4. El controlador redirecciona hacia una vista (JSP) cuya labor es crear una pantalla donde mostrara el resultado de la petición ejecutada. Los datos a desplegar le serán entregados por el controlador en un JavaBean.

5. La vista resultante se construye en un JSP, tomando los datos que requiera del Bean.

6. Finalmente se devuelve el resultado en una respuesta HTTP. En esta página el usuario vera el resultado de su petición y se tendrá opciones para realizar otras acciones, ejecutando una nueva petición por cada una de ellas, volviendo así a empezar el ciclo.

Beneficios:

- El modelo de datos se puede reutilizar en múltiples interfaces gráficas y subsistemas, puesto que no sabe nada del controlador o de la vista.

- Facilita la asignación de actividades entre los miembros del equipo de desarrollo, minimizando que el trabajo de uno interfiera con el de otro. Ej. Un diseñador gráfico se puede encargar de las vistas en HTML, mientras que un programador Java se puede encargar del controlador, mientras un tercero se encarga de los servicios en el modelo de datos.

De la descripción del patrón MVC, sabemos que se encuentra compuesto de tres patrones fundamentales, el Observer, el Strategy y el Composite, sin embargo estos patrones parecen no estar presentes en el “modelo 2”.

En cierta forma lo están, pero al estar adaptados a la idiosincrasia de la tecnología Web, tienen formas diferentes:

- Observer: La vista no es un Observer del modelo. Debido a que en el protocolo HTTP, después de cada petición, es necesario generar una nueva pantalla, es suficiente con que el controlador pase los nuevos datos a la vista para que genere una pantalla actualizada.

- Strategy: Podemos tener varios controladores (servlets) que podemos intercambiar al mandar a llamar el que nos provea el comportamiento requerido.

- Composite: Un JSP finalmente genera HTML que es dibujado por un navegador mediante una serie de etiquetas anidadas, tratando de forma “indistinta” a componentes visuales y a contenedores de otros componentes visuales.

V. CASO DE ESTUDIO: DESARROLLO DE UN FRAMEWORK PARA APLICACIONES WEB

INTRODUCCIÓN

En el desarrollo de distintas aplicaciones empresariales, suele presentarse de forma recurrente el tener que programar partes de la arquitectura que se asemejan mucho en los distintos proyectos.

Por citar algunos ejemplos:

- La forma de crear consultas y modificaciones a la base de datos (acceso a datos)
- El manejo de las transacciones en la base de datos
- La arquitectura que se adopte para la aplicación
- El manejo de la seguridad dentro de la aplicación (autenticación y autorización)
- La comunicación entre distintas partes del sistema
- La forma de crear interfaces gráficas para el usuario (GUI)

El objetivo de un framework es encapsular y proveer una o varias de estas funcionalidades desde una librería compartida de tal forma que no sea necesario codificarla en cada aplicación que se decida emprender.

Por ello el framework debe ser suficientemente flexible para manejar un número indeterminado de casos concretos, permitiendo al desarrollador configurarlo según las necesidades y naturaleza del proyecto.

OBJETIVO DEL FRAMEWORK

El objetivo del presente framework es ayudar al desarrollador a construir una aplicación web empresarial, particularmente en los siguientes aspectos:

- Implementar una **arquitectura** con una clara división de responsabilidades, tal que mejore la legibilidad y mantenimiento del código.
- Facilitar el **acceso a datos** contenidos en una base de datos relacional.
- Manejar la **transaccionalidad** en la ejecución de múltiples operaciones sobre la base de datos que permitan mantener la información en un estado consistente.

PATRONES DE DISEÑO APLICADOS

En el capítulo anterior se expusieron varios de los patrones más comunes, de la forma en que son encontrados en la literatura sobre patrones de diseño.

En el desarrollo de este framework se presentaron muchas oportunidades para utilizar estos patrones de diseño, y en esta sección se detallan algunos de los más sobresalientes, sin embargo cabe resaltar que varios de ellos no fueron implementados exactamente igual a lo indicado por la teoría en los textos.

Los patrones de diseño, al igual que los principios de diseño, son guías que reúnen la experiencia de mucha gente en múltiples proyectos y siempre se deben tomar en cuenta, pero cada ambiente de desarrollo es distinto y puede tener necesidades distintas, por lo que es posible modificar el uso de un patrón cuando se justifica, tomando en consideración las consecuencias que dicho cambio traerá sobre el código del sistema.

OPERACIONES SQL BINARIAS - TEMPLATE METHOD

Si se observa con detenimiento las condiciones más simples de SQL, la mayoría de ellas tienen la siguiente sintaxis:

```
<columna><operador><valor>
```

Esto implica que para construir las expresiones se seguirán los mismos pasos y solo hay pequeñas variaciones entre una condición y otra.

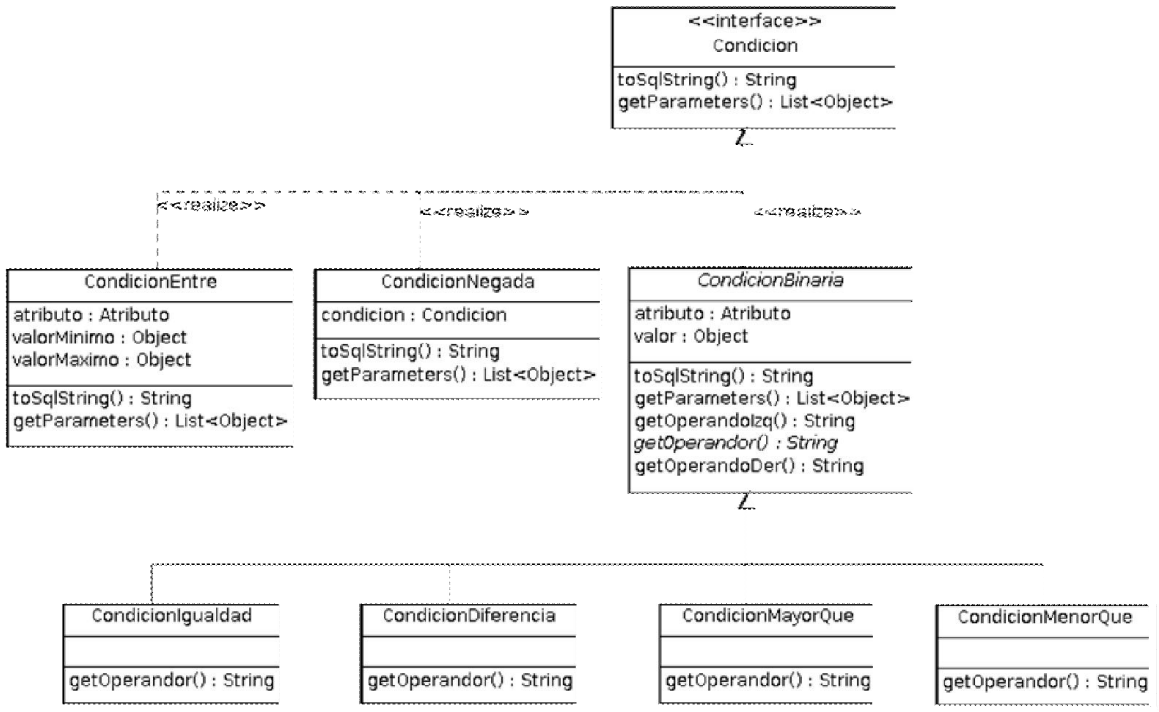

FIGURA V.1 - DIAGRAMA DE CLASES DE CONDICIONES BINARIAS

Es el patrón Template Method el que permite definir un algoritmo con una secuencia de pasos, pero permitir a las subclases definir algunos de esos pasos para diferenciarse del resto, por lo que se aplica aquí para simplificar la codificación de la mayoría de las condiciones simples.

La clase abstracta `CondicionBinaria`, define el algoritmo `toSqlString()` que consta de tres pasos:

1.- Concatenar el operando izquierdo (nombre de la columna a comparar)

2.- Concatenar el operador

3.- Concatenar el operando derecho (valor a comparar)

```
public abstract class CondicionBinaria implements Condicion {
    ...

    @Override
    public final String toSqlString() {
        String sql = getOperandoIzq();
        sql += getOperador();
        sql += getOperandoDer();

        return sql;
    }

    protected String getOperandoIzq() {
        return atributo.getColumna();
    }

    protected abstract String getOperador();

    protected String getOperandoDer() {
        return "?";
    }

    ...
}
```

Las condiciones binarias concretas heredan de `CondicionBinaria` y puesto que el algoritmo ya está definido en la clase padre, todo lo que las clases concretas deben hacer es sobrescribir el método `getOperador()` para especificar el operador a usar.

```
public class CondicionDiferencia extends CondicionBinaria {
    ...

    @Override
    protected String getOperador() {
        return "<>";
    }
}

public class CondicionMayorQue extends CondicionBinaria {
    ...

    @Override
```

```
    protected String getOperador() {
        return ">";
    }
}
```

Esto permite entonces crear condiciones sencillas, que delegan la mayor parte de la funcionalidad a la clase padre, y se usa mediante el siguiente código:

```
// Condicion para 'No desactivados'
Condicion condicionStatus = new CondicionDiferencia(status, "D");
String condicionSql = condicionStatus.toSqlString();

// Salida: status<>?
```

CONDICIONES SQL - COMPOSITE

En SQL se suele usar una condición (la cláusula WHERE) para limitar el resultado de una búsqueda y de esa forma solo afectar los registros que cumplan con una serie de criterios.

Esta condición puede ser simple como una comparación de equidad (Ej. nombre='Juan'), pero también puede ser compleja al combinar varios criterios simples con los operadores AND y OR (Ej. apellido='Moreno' AND (nombre='Pedro' OR nombre='Juan')).

Estos operadores (AND y OR) combinan 2 criterios simples para crear uno complejo, por lo que se puede decir que son una operación compuesta de otras más sencillas.

Pensarlo de esta manera nos permite crear estructuras en forma de árbol, que pueden crecer y hacerse tan complejas como se requiera, pero al mantener la uniformidad entre objetos simples y objetos compuestos la programación se puede simplificar significativamente.

Por ejemplo, de la siguiente sentencia SQL:

```
SELECT * FROM Persona WHERE nombre LIKE 'Juan%' AND (apellidoPaterno =
'Palacios' OR apellidoPaterno = 'Godinez') AND fechaNacimiento > '29-08-
1981';
```

Podríamos generar el siguiente diagrama de árbol con la sentencia WHERE:

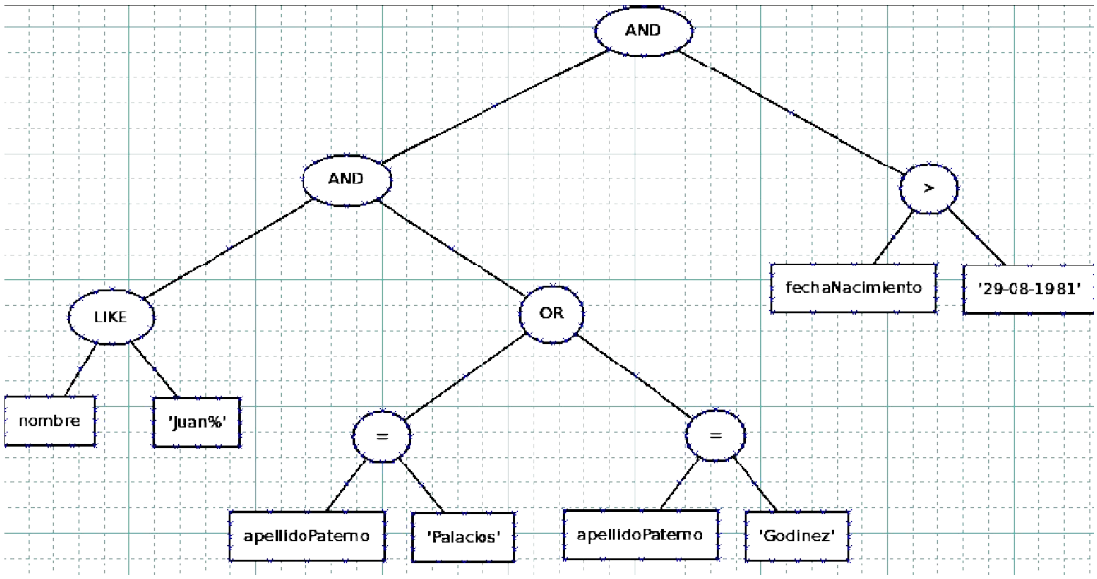

FIGURA V.2 - DIAGRAMA DE ÁRBOL DE UNA CONDICIÓN SQL

El patrón de diseño Composite, permite tratar a los elementos compuestos de la misma forma en que se trata a los elementos simples por lo que se puede usar aquí para simplificar el código al momento de crear y usar condiciones de SQL.

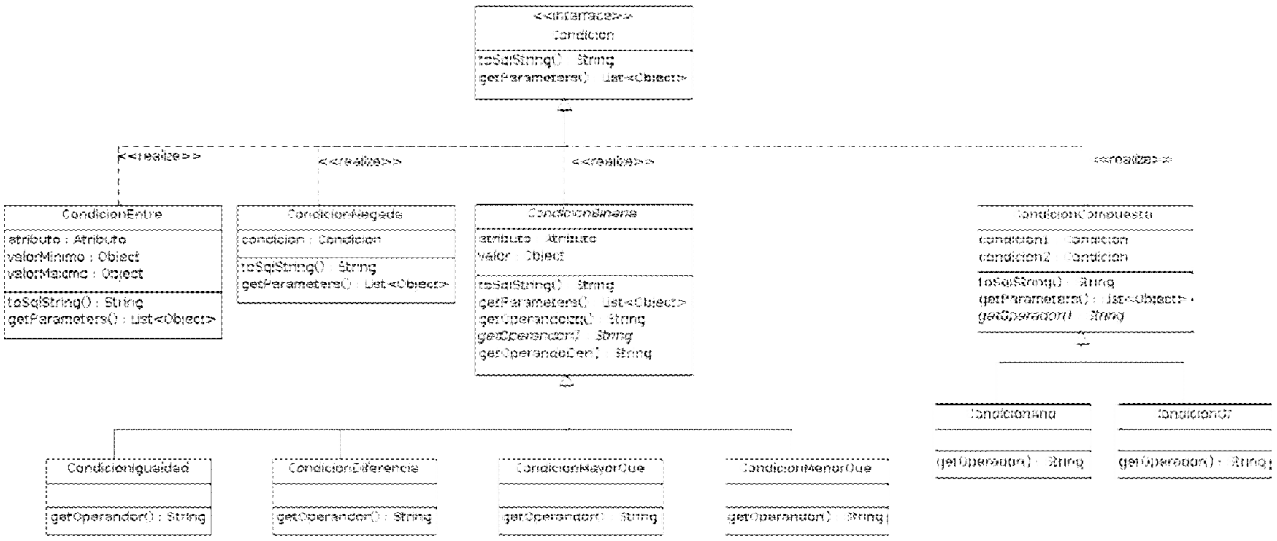

FIGURA V.3 - DIAGRAMA DE CLASES: CONDICIONES SQL

Las clases que implementan `Condición`, son las condiciones simples.

La clase abstracta `CondicionCompuesta` será la base de las condiciones compuestas por más de una condición simple. Implementa la interfaz `Condicion` tal que, transparentemente, sus subclases se pueden usar en cualquier lugar donde una `Condicion` sea apropiada.

```
public abstract class CondicionCompuesta implements Condicion {
    Condicion condicion1;
    Condicion condicion2;
```

```
    public CondicionCompuesta(Condicion condicion1, Condicion condicion2)
{
        this.condicion1 = condicion1;
        this.condicion2 = condicion2;
    }

    @Override
    public final String toSqlString() {
        String sql = "(" +
                condicion1.toSqlString() +
                getOperador() +
                condicion2.toSqlString() +
                ")";
        return sql;
    }

    public abstract String getOperador();

    @Override
    public List<Object> getParameters() {
        List<Object> parametros =
            new ArrayList<Object>(condicion1.getParameters());
        parametros.addAll(condicion2.getParameters());
        return parametros;
    }
}
```

Las clases `CondicionAnd` y `CondicionOr` son las composiciones concretas. Heredan de `CondicionCompuesta` por lo que implícitamente también heredan de la interfaz `Condicion`.

```
public class CondicionAnd extends CondicionCompuesta {
    public CondicionAnd(Condicion condicion1, Condicion condicion2) {
        super(condicion1, condicion2);
    }

    @Override
    public String getOperador() {
        return " AND ";
    }
}


public class CondicionOr extends CondicionCompuesta {
    public CondicionOr(Condicion condicion1, Condicion condicion2) {
        super(condicion1, condicion2);
    }

    @Override
    public String getOperador() {
        return " OR ";
    }
}
```

De esta manera, es posible combinar los distintos tipos de condiciones y representarlos mediante un objeto `Condicion` resultante.

Por ejemplo, continuando con la cláusula where de la sentecia SQL mencionada anteriormente, y si se recorre el arbol generado con un algoritmo de busqueda por profundidad (Depth-First Search), primero se crean las condiciones simples:

```
Condicion condicionNombre = new CondicionLike(nombre, "Juan%");
Condicion condicionApellido1 =
    new CondicionIgualdad(apellido, "Palacios");
Condicion condicionApellido2 =
    new CondicionIgualdad(apellido, "Godinez");
```

Después, se combinan mediante Condiciones Compuestas (AND y OR):

```
Condicion condicionApellido =
    new CondicionOr(condicionApellido1, condicionApellido2);
Condicion condicionNombreCompleto =
    new CondicionAnd(condicionNombre,  condicionApellido);
```

Y finalmente se combinan nuevamente con la condición de la fecha de nacimiento para obtener un solo objeto `Condicion`:

```
Calendar fecha = new GregorianCalendar(1981, 8, 29);
Condicion condicionNacimiento =
    new CondicionMayorQue(fechaNacimiento, fecha.getTime());

Condicion condicion =
    new CondicionAnd(condicionNombreCompleto,
          condicionNacimiento);
```

De esta manera, cuando se requiera generar la sentencia SQL, el contenido de la cláusula WHERE, se obtendrá haciendo uso del método `toSqlString()`, del objeto `Condicion` resultante:

```
String sqlWhere = condicion.toSqlString()
```

Método que orquestará a todas las condiciones simples y compuestas (por igual) contenidas en el objeto `Condicion` resultante para producir una sentencia SQL equivalente a la requerida. Para este ejemplo se genera la siguiente cadena:

```
"((nombre LIKE ? AND (apellidoPaterno=? OR apellidoPaterno=?)) AND
fechaNacimiento>?)"
```

FÁBRICA DE CONDICIONES - ABSTRACT FACTORY / FACTORY METHOD

Aún con el estándar de SQL-92, puede haber algunas diferencias en la sintaxis SQL de distintos manejadores de base de datos, e incluso podríamos llegar a usar una base de datos que no cumpla con el estándar.

Debido a ello no es deseable tener código como el siguiente en la aplicación:

```
Condicion condicion = new CondicionDiferencia(atributo, valor);
```

Ya que si se decidiera cambiar de manejador de base de datos, podría llegar a ser muy complicado si el nuevo manejador tiene una sintaxis diferente al previo.

Por ejemplo, en la mayoría de las bases de datos actuales, un símbolo de diferencia soportado es '!=', sin embargo en algunas bases de datos, éste no es soportado (por ejemplo en el manejador de base de datos MiniSQL, donde único operador de diferencia está dado por el símbolo '<>').

Eso implicaría crear una nueva clase de diferencia específica para MiniSQL que podríamos llamar `CondicionDiferenciaMSQL`.

Luego entonces, tendríamos que cambiar todas las condiciones de diferencia para usar la nueva clase:

```
Condicion condicion = new CondicionDiferenciaMSQL(atributo, valor);
```

Lo que podría convertirse en una pesadilla para el mantenimiento, por lo que es conveniente que el código cliente se desacople de todas la implementaciones concretas de la clase `Condicion`.

Para ello se usa una "*Fabrica Abstracta"* de condiciones, de tal forma que cada subclase represente la fábrica concreta a usar para una base de datos en específico.

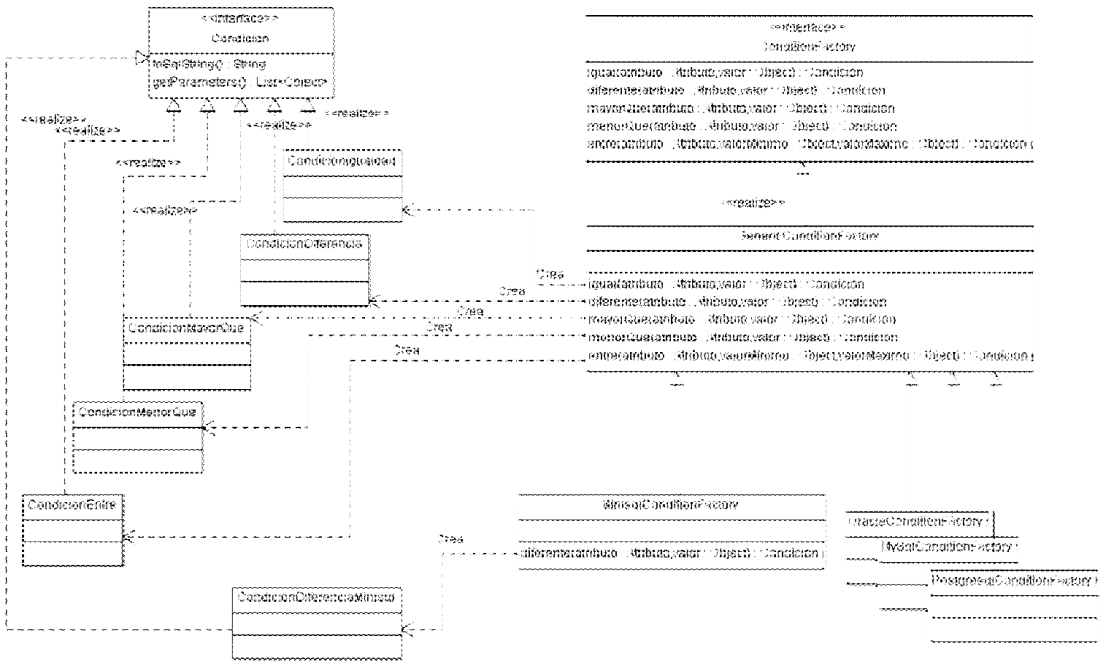

FIGURA V.4 - DIAGRAMA DE CLASES – FABRICAS DE CONDICIONES

`CondicionFactory`: Interface que define varios métodos para crear distintos tipos de condiciones SQL.

`GenericConditionFactory`: Puesto que en la actualidad la mayoría de los manejadores de bases de datos relacionales cumplen con el estándar SQL-92, se crea una implementación genérica de las condiciones SQL.

`MiniSqlConditionFactory`, `OracleConditionFactory`, etc.: Implementaciones para una base de datos particular (pueden heredar de `GenericConditionFactory` para no re-implementar todo aquello que tengan en común).

Cada Fábrica concreta es libre de instanciar la `Condicion` concreta que le sea apropiada, pero ésta permanecerá oculta del código cliente.

```
Condicion condicion = fabrica.diferente(atributo, valor);
```

Dependiendo de la configuración de la fábrica, ésta línea de código podría estar devolviendo la condición de diferencia de cualquier implementación posible (MSQL, Oracle, Sybase, etc.). En el caso planteado, podría ser la condición de diferencia genérica (`CondicionDiferencia`) o la condición de diferencia específica para MiniSQL (`CondicionDiferenciaMinisql`), con sólo configurar el uso de la fábrica concreta apropiada.

MANEJADOR DE CONEXIONES - SINGLETON

En una aplicación existen ciertos objetos de los cuales necesitamos que exista uno y sólo uno, ya que éste puede controlar y ser el punto de acceso a un recurso limitado. Los accesos a una base de datos es un ejemplo de un recurso limitado que se debe acceder a través de un único punto, a fin de controlar el número de conexiones ya que éstas son costosas de crear y tener un número elevado de conexiones simultáneas puede degradar el tiempo de respuesta de la base de datos.

El objeto `PoolConexiones`, tiene la funcionalidad de administrar las conexiones a la base de datos (esto es crear, preparar, reciclar y destruir las conexiones cuando se considere apropiado) y puede ser accedido desde distintas partes de la aplicación, por lo que es un candidato ideal para aplicar el patrón `Singleton` sobre él.

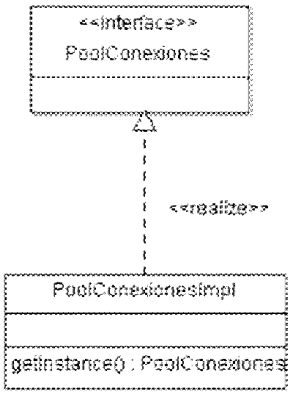

FIGURA V.5 - DIAGRAMA DE CLASES: POOLCONEXIONES SINGLETON

Puesto que es un objeto que es "buscado" constantemente y en un ambiente multihilos (ya que una aplicación empresarial se suele levantar un hilo para atender a cada usuario simultáneo en el sistema), es importante que sea "thread-save" y también no degradar el performance con métodos sincronizados excesivamente, por lo que en este caso se implementó este singleton con la técnica de "Double Checking".

```
public class PoolConexionesImpl implements PoolConexiones {
    private static PoolConexionesImpl instance;
    ...
    private PoolConexionesImpl() throws SQLException {
    }

    public static PoolConexionesImpl getInstance() throws SQLException {
        if(instance == null) {
            instance = createInstance();
        }

        return instance;
    }

    private static synchronized PoolConexionesImpl createInstance()
                                                    throws SQLException {
        if(instance == null) {
            instance = new PoolConexionesImpl();
        }

        return instance;
    }
}
```

Así la forma de obtener una referencia al pool de conexiones es mediante el método `getInstance()`, que es el responsable de devolver **siempre** la misma instancia:

```
PoolConexiones pool = PoolConexionesImpl.getInstance();
```

MANEJADOR DE CONEXIONES A BASE DE DATOS - OBJECT POOL

Las conexiones a base de datos son muy costosas de hacer (toma tiempo, consume muchos ciclos de procesador, operaciones de entrada/salida, etc.) y las bases de datos suelen tener un límite de conexiones abiertas ya que el desempeño de la base decrece si tienen que manejar demasiadas de forma concurrente.

Además en una aplicación empresarial se suele ocupar la conexión por muy poco tiempo (para realizar una consulta) para después destruir el thread que la estaba ocupando.

El patrón "Object Pool" es un patrón creacional que utiliza un conjunto de objetos inicializados y listos para ser usados, en vez de crearlos y destruirlos cuando son requeridos, lo que hace que sea el ideal para diseñar un manejador de conexiones.

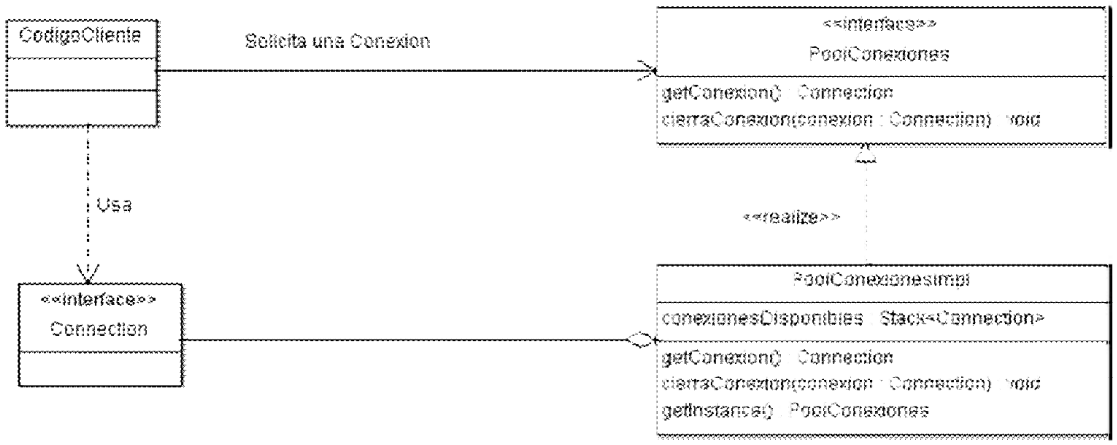

FIGURA V.6 - DIAGRAMA DE CLASES DE POOLCONEXIONES

`CodigoCliente`: Es todo aquel código de la aplicación desde donde se acceda a la base de datos para realizar consultas o modificaciones.

```
//Se obtiene una referencia al pool
PoolConexiones pool = PoolConexionesImpl.getInstance();
//Del pool se obtiene una conexión disponible
Connection conexion = pool.getConexion();
// Se usa la conexión, para realizar una consulta cualquiera.
conexión.prepareStatement("SELECT * FROM Persona where id=?")
...
// Y se devuelve la conexión al pool para su reutilización
pool.cierraConexion(conexion);
```

`Connection` (Reusable): Las conexiones a la base de datos son los objetos administrados por el Pool, lo que permitirá reutilizar dichas conexiones.

`PoolConexiones` (ReusablePool): Es el administrador de las conexiones. Esencialmente, define un método que el cliente usa para obtener una conexión a la base de datos (`getConexion`) y otro para indicar que ha terminado de usarla y que puede ser devuelta al pool de conexiones disponibles (`cierraConexion`).

```
public interface PoolConexiones {
    Connection getConexion() throws SQLException;
    void cierraConexion(Connection connection);
    void iniciaPool() throws SQLException;
    void destruyePool();
}
```

`PoolConexionesImpl`: Es a una posible forma de implementar el Pool de Conexiones. Esta clase define el comportamiento real del Pool, al que se le pueden agregar otras características, según se requieran, como puede ser, un número inicial de conexiones a crear, un número máximo de conexiones simultáneas creadas, un tiempo máximo de espera para regresar una conexión al pool (en caso de que el cliente no devuelva la conexión), etc..

```
public class PoolConexionesImpl implements PoolConexiones {
    private static PoolConexionesImpl instance;

    private List<Connection> conexionesDisponibles;
    private Map<Connection, Long> conexionesOcupadas;

    private Thread hiloValidador;
    private boolean running;
    private ConfiguracionPoolBD config;

    private PoolConexionesImpl() throws SQLException {
    }

    public static PoolConexionesImpl getInstance() throws SQLException {
        if(instance == null) {
            instance = createInstance();
        }

        return instance;
    }

    public static synchronized PoolConexionesImpl createInstance()
                                                    throws SQLException {
        if(instance == null) {
            instance = new PoolConexionesImpl();
        }

        return instance;
    }
    // Crea una nueva conexión a la base de datos
    private Connection creaConexion() throws SQLException {
        Connection conexion = null;

        String url = config.getUrl(); //"jdbc:mysql://localhost:3306/"
        String dbName = config.getNombreBD();
        String driver = config.getDriver(); //"com.mysql.jdbc.Driver"
        String userName = config.getUsuario();
        String password = config.getContrasena();
        try {
            Class.forName(driver).newInstance();
            conexion = DriverManager.getConnection(url + dbName,
                  userName, password);
            conexion.setAutoCommit(false);
            System.out.println("Conexion Creada: " +
                  conexion.toString());
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
```

```
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }

        return conexion;
    }

    @Override
    public synchronized Connection getConexion() throws SQLException {
        Connection conexion;
        // Si hay conexiones disponibles
        if(!conexionesDisponibles.isEmpty()) {
            // Se toma la primera
            conexion = conexionesDisponibles.remove(0);
        } else {
            // Se crea una nueva temporalmente para atender la peticion
            conexion = creaConexion();
        }

        // Registra la conexion como Ocupada (No disponible)
        conexionesOcupadas.put(conexion, System.currentTimeMillis());
        return  conexion;
    }

    @Override
    public synchronized void cierraConexion(Connection conexion) {
        // Se elimina de la lista de conexiones ocupadas
        conexionesOcupadas.remove(conexion);

        try {
            // Si hay espacio en el pool
            if(conexionesDisponibles.size() < config.getTamanoPool()) {
                // Se devuelve la conexion

                // Si la conexión aún es valida
                if(validaConexion(conexion)) {
                    // Se reutiliza
                    conexionesDisponibles.add(conexion);
                } else {
                    // Se crea una nueva
                    conexionesDisponibles.add(creaConexion());
                }
            } else {
                // Se destruye la conexion para liberar recursos
                if(validaConexion(conexion)) {
                    conexion.close();
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void iniciaPool(ConfiguracionPoolBD config)
             throws SQLException {
        this.config = config;
```

```
        conexionesDisponibles =
            new ArrayList<Connection>(config.getTamanoPool());
        conexionesOcupadas =
            new HashMap<Connection, Long>(config.getTamanoPool());

        for(int i=0; i<config.getTamanoPool(); i++) {
            Connection conexion = creaConexion();
            conexionesDisponibles.add(conexion);
        }

        hiloValidador = new Thread(new ValidaCaducidadRunnable());
        running = true;
        hiloValidador.start();
    }

    @Override
    public void destruyePool() {
        running = false;
        hiloValidador.interrupt();
    }

    private boolean validaConexion(Connection conexion) {
        try {
            return conexion.isClosed();
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return false;
    }

    // Thread que valida constantemente si las conexiones han caducado
    // En cuyo caso, las devuelve automáticamente al pool
    private class ValidaCaducidadRunnable implements Runnable {
        @Override
        public void run() {
            System.out.println("Iniciando la validacion: " + running);
            while(running) {
                try {
                    Thread.sleep(config.getTiempoEspera());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println("Validando si hay por expirar");

                for (Connection conexion : conexionesOcupadas.keySet()) {
                    Long horaExpedicion =
                        conexionesOcupadas.get(conexion);
                    // Si se ha superado el tiempo limite
                    if(System.currentTimeMillis() - horaExpedicion >
                        config.getTiempoEspera()) {
                        // Se regresa la conexion al pool
                        System.out.println("Cerrando conexion Caduca");
                        cierraConexion(conexion);
                    }
                }
```

```
            }

            // Libera Recursos

            // Libera Conexiones Disponibles
            Iterator<Connection> iterador =
                  conexionesDisponibles.iterator();
            while (iterador.hasNext()) {
                Connection conexion = iterador.next();
                try {
                    conexion.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
                iterador.remove();
            }

            // Libera Conexiones Ocupadas
            for (Connection conexion : conexionesOcupadas.keySet()) {
                try {
                    conexion.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
                conexionesOcupadas.remove(conexion);
            }
        }
    }
}
```

FÁBRICA DE BEANS - MÉTODO SIMPLE DE FABRICACIÓN DINÁMICO

Partiendo del concepto de un “Método Simple de Fabricación”, pero a fin de evitar una larga lista de condicionantes y objetos creados con el operador new, se recurrió a usar el API "java reflection" para la creación de las instancias.

Lo normal en un patrón "Fábrica" es que el método devuelva la interfaz que implementen todos los objetos de la familia que esa fábrica puede crear, sin embargo, para esta fábrica en especial, esa familia son todos los objetos Bean del modelo de datos, lo que implica que serán una buena cantidad de objetos, y que además estos objetos son importantes para el cliente que los usa por sus particularidades (i.e. requiere el objeto concreto, no la interfaz), por lo que se modificó el patrón, usando generics de java 5.0, para que devuelva directamente el mismo tipo de objeto solicitado.

La firma del método tiene la siguiente forma:

```
public class BeanFactory {
      public <T extends BaseBean> T getBean(Class<T> tipoBean) {
        try {
            T bean = tipoBean.newInstance();
            return bean;
        } catch (IllegalArgumentException e) {
            throw new RuntimeException(
            "Argumentos equivocados para instanciar "+
```

```
                tipoBean.getName(), e);
            } catch (InstantiationException e) {
                throw new RuntimeException(
                "No se pudo instanciar "+
                tipoBean.getName(), e);
            } catch (IllegalAccessException e) {
                throw new RuntimeException(
                "No se tiene permiso para instanciar "+
                tipoBean.getName(), e);
            }
        }
}
```

Con esto se logra encapsular en un método la instanciación de todos los objetos del modelo de datos, siendo esto especialmente importante para los objetos que se deben crear dinámicamente dentro del framework para generar una respuesta a una petición de datos.

Instanciar un objeto mediante este método requiere indicar el tipo de objeto a instanciar:

```
BeanFactory factory = new BeanFactory();
Usuario usuario = factory.getBean(Usuario.class);
```

TRANSACCIONALIDAD EN LOS SERVICIOS - PROXY DINÁMICO

Cada operación que se pueda realizar sobre el modelo de datos, puede estar compuesta de varias órdenes (sentencias) que se ejecutan de forma individual sobre la base de datos.

Un conjunto de órdenes que se ejecutan sobre una base de datos formando una unidad de trabajo, de forma indivisible o atómica, se conoce como una transacción.

Esto nos dice que cada operación que se realice sobre el modelo de datos debe ocurrir en una sola transacción para mantener la integridad y congruencia de los datos. Si alguna de las sentencias de la transacción falla, la base de datos debe regresar al estado en que se encontraba al iniciar la transacción.

Ejemplo de operación transaccional: Transferencia Bancaria

En un banco, cuando se realiza una transferencia de fondos (por un monto X) entre dos cuentas (A y B) básicamente se realizan dos operaciones:

- Se resta un monto X del saldo de la cuenta A (retiro)
- Se suma un monto X al saldo de la cuenta B (deposito)

Es de suma importancia que al realizar la transferencia se asegure que ambas operaciones se hayan realizado correctamente, puesto que de otra forma se crearían inconsistencias en la contabilidad del banco.

Si sólo la primera operación fuese ejecutada, mientras que la segunda fallara, se perdería del sistema el monto X retirado de la cuenta A. Por otro lado, si sólo se ejecutara la segunda operación, mientras que fallara la primera, entonces entraría de la nada un monto X al sistema, que sería depositado en la cuenta B.

El API Java de conexión a base de datos "JDBC", provee un mecanismo para el manejo de operaciones transaccionales. Éste consiste en que el programador inicie una transacción guardando el estado actual de la base de datos con el método `setSavepoint()`, luego ejecuta las sentencias individuales requeridas para esa transacción y finalmente guarda de forma permanente el nuevo estado de la base de datos con el método `commit()`. Si alguna de las sentencias individuales no se pudiera ejecutar correctamente, el programador utiliza el método `rollback()` para cancelar cualquier cambio que se pudiera haber hecho y regresar al estado previo.

Ejemplo java de operación transaccional:

```
public void transfiere(String cuentaOrigen, String cuentaDestino,
                       BigDecimal monto) throws Throwable {
      try {
            // Antes de modificar algo
            // Guarda el estado actual de la bd
            con.setSavepoint();

            retira(cuentaOrigen, monto);
            deposita(cuentaDestino, monto);

            // Después de ejecutadas las operaciones
            // Guarda el nuevo estado de la bd
            con.commit();
      } catch (Exception e) {
            // En caso de ocurrir un error en el retiro o en el deposito
            // Se regresa la bd al estado inicial
            con.rollback();

            throw e;
      }
}
```

Del ejemplo, es notorio que cada operación que modifique valores en la base de datos requiere repetir el mismo procedimiento para asegurar la transaccionalidad, lo que en el código fuente se ve reflejado al repetir las sentencias `setSavepoint()`, `commit()` y `rollback()` en cada operación transaccional.

Para evitar "contaminar" el código del servicio, que contiene la lógica de negocio, con el código repetitivo que maneja la transaccionalidad es posible recurrir a utilizar un "Proxy", que envuelva al servicio de tal forma que inicie una transacción antes de entrar al método que ejecuta la operación, y que una vez terminada la ejecución, termine la transacción y que en caso de algún error, regrese a la base de datos al estado anterior.

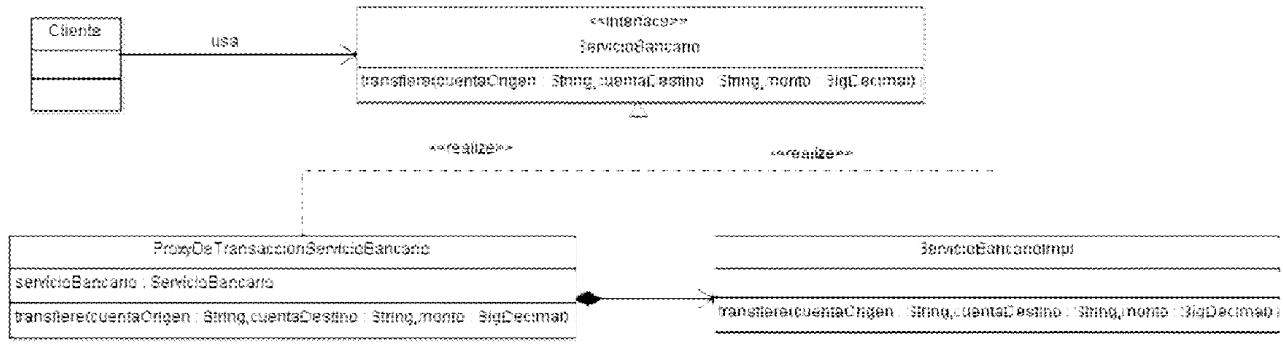

FIGURA V.7 - DIAGRAMA DE CLASES: PROXY TRANSACCIONAL PARA UN SERVICIO BANCARIO

Ejemplo de proxy que maneja la transaccionalidad de un servicio:

```
public class ProxyDeTransaccionServicioBancario implements
ServicioBancario {
    private ServicioBancario servicio;

    public ProxyDeTransaccionServicioBancario(ServicioBancario servicio)
{
        this.servicio = servicio;
    }

    @Override
    public void transfiere(String cuentaOrigen, String cuentaDestino,
                           BigDecimal monto) throws Throwable {
        SqlConnector connector = SqlConnector.getInstance();
        Connection connection = connector.getConnection();

        try {
            // Antes de modificar algo
            // Guarda el estado actual de la bd
            connection.setSavepoint();

            // Se invoca el servicio real
            servicio.transfiere(cuentaOrigen, cuentaDestino, monto);

            // Después de ejecutadas las operaciones
            // Guarda el nuevo estado de la bd
            connection.commit();
        } catch (Exception e) {
            // En caso de ocurrir un error en el retiro o en el deposito
            // Se regresa la bd al estado inicial
            connection.rollback();

            throw e;
        } finally {
            // No olvidemos liberar la conexion
            connector.closeConnection(connection);
        }
    }
}
```

Ya que se ha movido todo el código que maneja la transaccionalidad al Proxy, el código del servicio es mucho más claro y conciso:

```
public class ServicioBancarioImpl implements ServicioBancario {
    @Override
    public void transfiere(String cuentaOrigen, String cuentaDestino,
                           BigDecimal monto) throws Throwable {
        retira(cuentaOrigen, monto);
        deposita(cuentaDestino, monto);
    }
...
}
```

Y su uso, sólo requiere instanciar el servicio y embeberlo dentro del proxy, que para todo fin, actuara como fachada para acceder al servicio:

```
ServicioBancario servicioBancario =
      new ProxyDeTransaccionServicioBancario(new ServicioBancarioImpl());
servicioBancario.transfiere("A", "B", BigDecimal.ONE);
```

A pesar de que el patrón Proxy logra separar la lógica de negocio en el servicio del código que maneja la transaccionalidad, es posible ver que por sí sólo no eliminará el problema de repetir la lógica para manejar la transacción, ya que si se agregan más operaciones al servicio, habrá que copiar y pegar el código de transaccionalidad del proxy en cada operación. Y más aún, si se agregan servicios nuevos al sistema, se requiere también crear un proxy, muy similar al mostrado pero que ahora implementará la interfaz de los nuevos servicios.

Sería posible combinar patrones adicionales (como el Template Pattern) para lograr separar y reutilizar la lógica que no varía al manejar las transacciones en todas las operaciones, sin embargo esto no evitaría tener que codificar nuevos objetos Proxy y tener que agregar a éstos cada operación nueva que se agregue al servicio.

Es por ello que se prefirió modificar el patrón Proxy tradicional, para dar pie a generar uno que dinámicamente en tiempo de ejecución implemente la interfaz del servicio para el que es requerido.

Para lograr un Proxy dinámico se recurrió al API de Java “Reflection”, encargado de introspectar y obtener información de las clases y objetos, además de permitir crear objetos dinámicamente en tiempo de ejecución.

```
public class ProxySqlTransaccional implements InvocationHandler {
    private ServicioTransaccional implementation;
    private SqlConnector connector;

    public ProxySqlTransaccional(ServicioTransaccional servicio,
                                 SqlConnector connector) {
        this.implementation = servicio;
        this.connector = connector;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        Object resultado;
```

```
        Connection connection = connector.getConnection();

        try {
            // Antes de modificar algo
            // Guarda el estado actual de la bd
            connection.setSavepoint();

            implementation.setConexion(connection);

            // Se invoca el servicio real
            resultado = method.invoke(implementation, args);

            // Después de ejecutadas las operaciones
            // Guarda el nuevo estado de la bd
            connection.commit();
        } catch (Exception e) {
            // En caso de ocurrir un error en el retiro o en el deposito
            // Se regresa la bd al estado inicial
            connection.rollback();

            throw e;
        } finally {
            // No olvidemos liberar la conexion
            connector.closeConnection(connection);
        }

        return resultado;
    }

    public static <T> T creaProxySqlTransaccional(Class<T> serviceType,
                                ServicioTransaccional servicio,
                                SqlConnector connector) {
        Class[] interfaces = {serviceType};
        T instancia = (T) creaProxyTransaccional(servicio,
                            interfaces, connector);

        return instancia;
    }

    private static Object creaProxyTransaccional(
                        ServicioTransaccional obj,
                        Class[] interfaces,
                        SqlConnector connector) {
        return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                interfaces,
                new ProxySqlTransaccional(obj, connector));
    }
}
```

Bajo este esquema, la implementación del servicio cambia ligeramente, debiendo implementar la interfaz `ServicioTransaccional`:

```
public class ServicioBancarioImpl implements ServicioBancario,
ServicioTransaccional {
    private ThreadLocal<Connection> conexion =
                        new ThreadLocal<Connection>();
```

```
    @Override
    public void setConexion(Connection conexion) {
        this.conexion.set(conexion);
    }

    public Connection getConexion() {
        return conexion.get();
    }

    @Override
    public void transfiere(String cuentaOrigen, String cuentaDestino,
                           BigDecimal monto) throws Throwable {
        retira(cuentaOrigen, monto);
        deposita(cuentaDestino, monto);
    }
...
}
```

Para el uso de este proxy, se agregó al proxy un método estático para crear instancias del propio proxy, que recibe la interfaz del servicio a implementar dinámicamente, el propio servicio (que debe implementar la interfaz `ServicioTransaccional` para poder recibir la conexión a la base de datos sobre la que debe realizar las operaciones transaccionales) y el conector de donde se obtendrán las conexiones a la base de datos.

```
ServicioBancario servicioBancario =
      ProxySqlTransaccional.creaProxySqlTransaccional(
            ServicioBancario.class,
            new ServicioBancarioImpl(),
            new SqlConnectorDemo(null));
servicioBancario.transfiere("A", "B", BigDecimal.ONE);
```

Salida de transferir $1.00 de una cuenta “A” a una cuenta “B”, ambas con $10.00 inicialmente:

```
Saldo de la cuenta actualizado: 9
Se han retirado 1 de la cuenta A

Saldo de la cuenta actualizado: 11
Se han abonado 1 a la cuenta B
```

DIVISIÓN DE RESPONSABILIDADES - MODEL 2

En una aplicación web de dimensiones medianas a grandes, es conveniente separar las responsabilidades de los diferentes componentes del sistema en capas bien definidas, de tal forma que reduzca el número de responsabilidades de cada componente y así facilite el mantenimiento y aprovechar gente especializada en un área (diseñadores, programadores de servicios y programadores de interfaces gráficas).

Para aplicaciones web creadas con tecnología Java, el Modelo 2 ofrece una forma de separar las responsabilidades, similar al modelo MVC ampliamente usado en diversas tecnologías.

Cuando se empieza a programar con JSP's y Servlets es difícil entender cómo implementar esta separación de forma efectiva. Es por ello que puede ser útil que el framework establezca las bases para implementar dicha arquitectura.

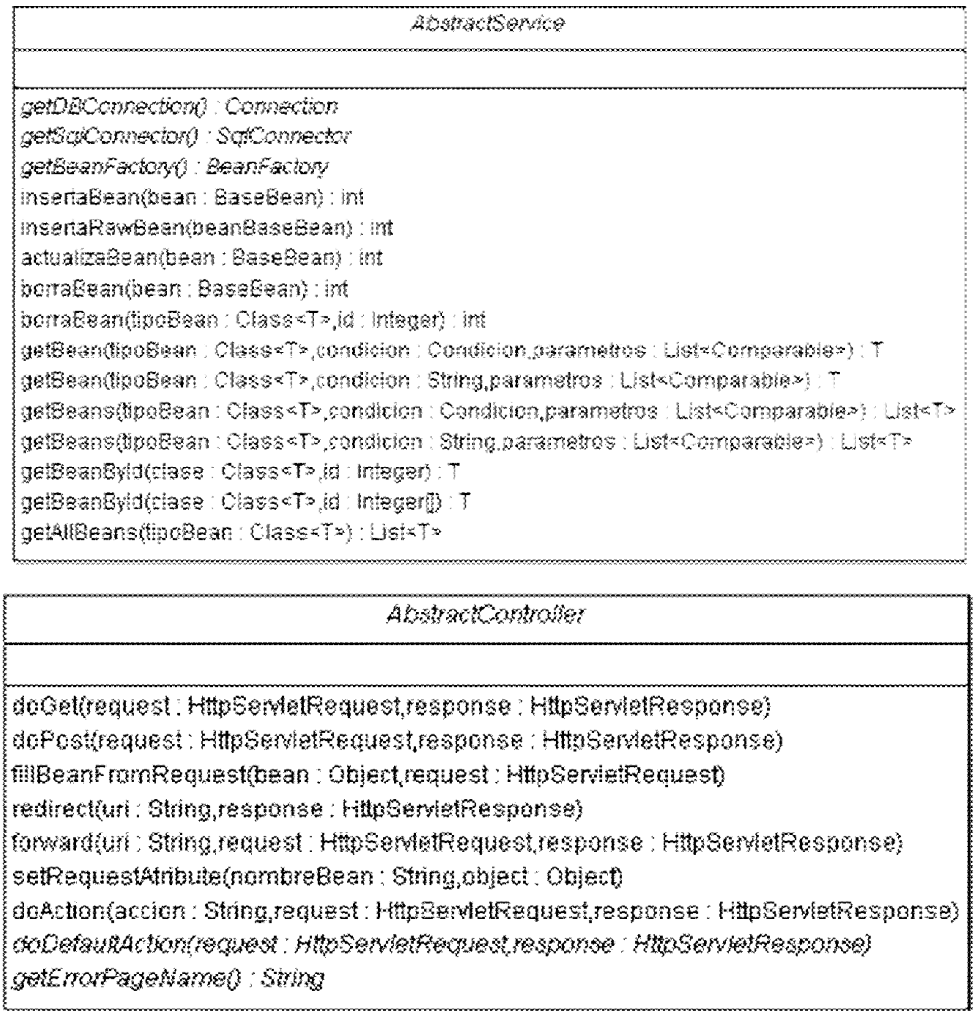

FIGURA V.8 - DIAGRAMA DE CLASES: MODEL 2

- Modelo: Una capa de servicios puede servir de interfaz para interactuar con los datos, proveyendo de operaciones para realizar consultas y otras para realizar y guardar modificaciones a los mismos.

  La clase `AbstractService` proporciona una clase base de la que los servicios concretos del Modelo de la aplicación pueden extender para heredar funcionalidad que les permita realizar operaciones básicas con la base de datos, incluyendo crear registros nuevos (`insertaBean`), actualizar registros existentes (`actualizaBean`), eliminar registros (`borraBean`) y realizar consultas (`getBean`, `getBeanById`, `getBeans` y `getAllBeans`).

- Controlador: Los controladores reciben solicitudes http del usuario y contienen lógica que analiza los datos recibidos e interpreta la acción que el usuario desea

realizar, por lo que para facilitar esta tarea se diseñó la clase `AbstractController`, que provee de herramientas que facilitan trabajar con parámetros http, crear beans para almacenar datos que se van a desplegar y finalmente redirigir hacia una vista (JSP).

El `AbstractController` fue diseñado para atender acciones del usuario, recibe un parámetro "action=xxxx" con cuyo valor invoca un método con la forma doXxxx() gracias a la implementación que provee para los métodos `doGet` y `doPost` de los `Servlets`.

Para facilitar trabajar con los datos de una petición http, se incluyó el método `fillBeanFromRequest`(), que almacena en un objeto los parámetros que se proporcionen.

Si la acción solicitada consulta o modifica información, entonces el controlador accede a estos mediante una capa de servicios del Modelo de datos.

Después de procesar la petición del usuario, es común que el controlador determine la siguiente pantalla a mostrar, por lo que se usa el método `setRequestAttribute`() para almacenar, de forma temporal, los datos que la siguiente pantalla requiere y se redirecciona la petición (mediante los métodos `redirect`() y `forward`()) hacia una vista.

- Vista: Las distintas pantallas html son generadas por Java Server Pages, que son invocados desde un controlador y que obtienen los datos a desplegar desde beans (llenados previamente por el mismo controlador a partir de datos obtenidos del modelo). Esto hace que el JSP ignore por completo el origen de los datos y se concentre en desplegarlos de forma correcta, limitando así sus responsabilidades y por tanto simplificando su implementación.

Así entonces, al usar el framework en una aplicación, se podría crear una jerarquía de controladores y servicios que hereden de `AbstractController` y `AbstractService`, respectivamente.

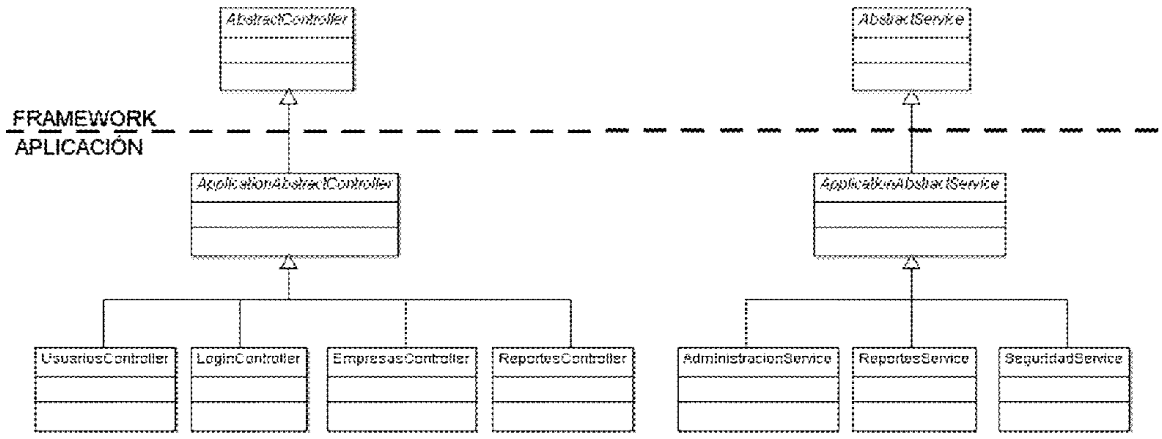

FIGURA V.9 - JERARQUÍA DE CONTROLADORES Y SERVICIOS EN UNA APLICACIÓN

A su vez, cada controlador concreto, podría administrar una o varias vistas relacionadas entre sí.

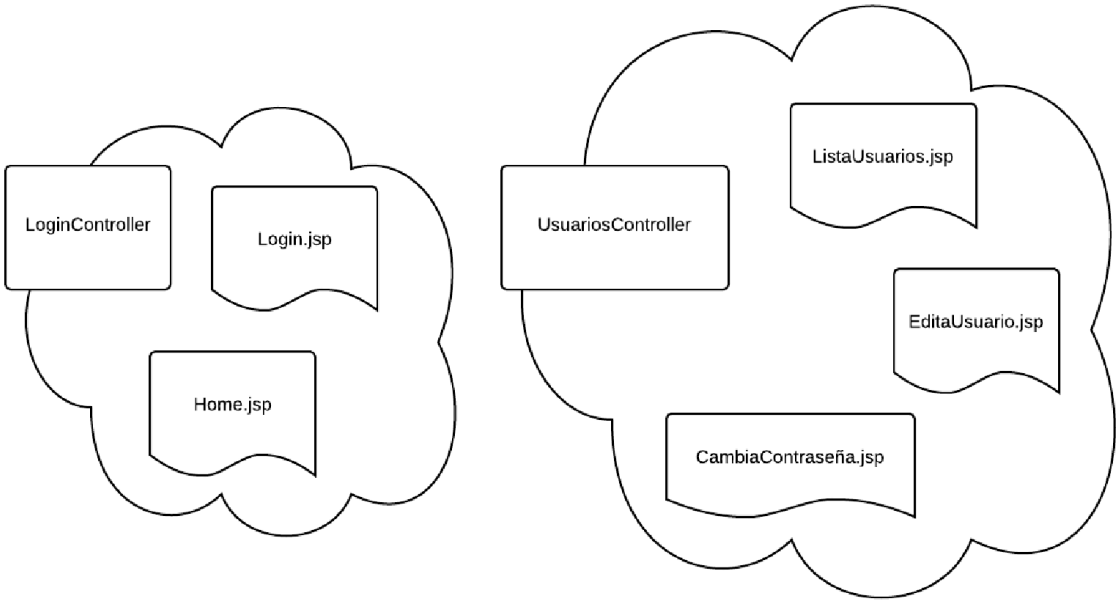

FIGURA V.10 - RELACIÓN CONTROLADOR / VISTAS

Por ejemplo, `LoginController` se puede encargar de presentar la pantalla Login.jsp donde se solicitarán al usuario sus credenciales para ingresar al sistema. Cuando el usuario realice una acción en la pantalla, se realizará una petición al `LoginController` que recibirá los datos introducidos y hará uso del `SeguridadService` para autenticar al usuario. Dependiendo del resultado, podrá elegir entre desplegar nuevamente la pantalla de Login.jsp con un mensaje de error, o re-direccionar hacia `Home.jsp` que presentará la pantalla principal de la aplicación.

RESUMEN

El framework construido proporciona al programador un marco de trabajo con varias herramientas útiles en la creación de un sistema empresarial web.

Al implementar parcialmente el patrón “Model 2”, invita al programdor a separar la implementación del sistema en los componentes de modelo, vista y controlador, lo que impone una estructura conocida y ordenada, lo que a la larga, facilita la compresión y el mantenimiento del mismo.

El componente del modelo, es el que hizo uso de una variedad de patrones de diseño para simplificar la interacción con la base de datos, intentando evitar escribir directamente sentencias sql y mantener un lenguaje orientado objetos.

Por ejemplo, en vez de escribir una sentencia SQL para realizar una inserción en la base de datos, se crea un objeto con los datos a guardar en el registro y se invoca el método `insertaBean()` proporcionado por `AbstractService`.

```
Usuario usuario = new Usuario(“shoguren”, “Cuauhtemoc Hohman”, 1, true);
insertaBean(usuario);
```

Por otro lado, si lo que se desea es realizar una consulta, es posible hacerlo definiendo los criterios de búsqueda en un objeto `Condicion` (ya sea simple o compuesta) e invocar el método `getBean()` y `getBeans()`.

```
Condicion condicionUsuario =
      condicionFactory.igual(Usuario.NOMBRE_USUARIO, "shoguren");
Usuario usuario = getBean(Usuario.class, condicionUsuario);
```

Las actualizaciones se pueden realizar sobre el objeto obtenido de la base de datos y se manda persistir con el método `actualizaBean()`.

```
usuario.setHabilitado(false);
actualizaBean(usuario);
```

Y para borrar un registro se puede ejecutar el método `borraBean()`.

```
borraBean(usuario);
```

Los servicios del modelo pueden realizar todas sus operaciones dentro de una sola transacción si implementan la interfaz `ServicioTransaccional` y se usa el proxy dinámico `ProxySqlTransaccional` como envoltorio del servicio, lo que evitar mezclar la lógica transaccional con la lógica del negocio.

```
public class ServicioSeguridadImpl
            implements ServicioSeguridad, ServicioTransaccional {

    @Override
    public Sesion autenticaUsuario(String usuario, String pwd)
                            throws AutenticacionFallida {
        CondicionFactory condicionFactory =
                  new GenericConditionFactory();

        pwd = encriptaContraseña(pwd);
```

```
        Condicion condicion = condicionFactory.and(
            condicionFactory.igual(Usuario.NOMBRE_USUARIO, usuario),
            condicionFactory.igual(Usuario.CONTRASENA, pwd)
        );

        Usuario usuario = getBean(Usuario.class, condicion);
        if(usuario == null) {
            throw new AutenticacionFallida("Nombre de usuario o
contraseña incorrecta");
        }

        registraEntrada(usuario);

        Sesion nuevaSesion = creaSesion(usuario);

        return nuevaSesion;
    }
...
```

Instanciando el servicio de seguridad envuelto con un proxy dinámico:

```
ServicioSeguridad servicioSeguridad =
      ProxySqlTransaccional.creaProxySqlTransaccional(
            ServicioSeguridad.class,
            new ServicioSeguridadImpl(),
            sqlConnector);
```

Algún controlador puede hacer uso de los servicios del modelo y la clase `AbstractController`, proporciona varios métodos útiles para leer los datos proporcionados en una petición HTTP y llenar con ellos objetos java, invocar servicios del Modelo para realizar búsquedas o modificar registros, guardar beans de información en la petición para finalmente invocar una vista que mostrará los resultados.

```
public class Login extends AbstractController {

    public void doDefaultAction(HttpServletRequest request,
HttpServletResponse response) {
      log.info("Ejecutando Login");
      // Redirecciona a la pantalla de autenticación
      forward("seguridad/login.jsp",request,response);
    }

    public void doEntra(HttpServletRequest request, HttpServletResponse
response) {
      log.info("Intentando entrar");
      String login = request.getParameter("login");
      String passwd = request.getParameter("passwd");

      try {
          Sesion sesion = getSeguridadService().
                        autenticaUsuario(login, passwd);

          session.setAttribute("sesion", sesion);

          //Lo mandamos a la pagina pricipal
          redirect(MENU_PAGE, response);
```

```
        } catch (AutenticacionFallida e) {
          //Si no existe, redirigirlo a la pagina de login con un error
          setRequestAtribute("mensajeError", e.getMessage());
          redirect(LOGIN_PAGE, response);
        }
    }
...
```

En la lógica del controlador mostrado, el usuario es redireccionado por defecto a la vista jsp login.jsp, donde se solicitan nombre de usuario y contraseña para ingresar al sistema.

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"
errorPage="errorPage.jsp" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>Entrada al Sistema InfoManejo</title>
    <link rel="stylesheet" href="css/ShortStyle.css" type="text/css"/>
</head>
<body>
<jsp:include page="/menu.jsp">
    <jsp:param name="modulo" value="Sistema InfoManejo"/>
</jsp:include>
<% String mensajeError = request.getParameter("mensajeError"); %>
<form method="post" action="Login">
    <input type="hidden" name="accion" value="entra"/>
    Nombre de Usuario: <input type="text" name="login"/><br/>
    Contraseña: <input type="password" name="passwd"/><br/>
    <input type="submit" value="Entrar">
</form>
<p style="error">
    <%=mensajeError != null ? mensajeError : ""%>
</p>
</body>
</html>
```

Cuando el usuario llena los datos solicitados en el formulario y da click en el botón “Entrar”, se envía una petición HTTP al controlador Login y se ejecuta la acción “entra” que invoca al método `doEntra`() de ese controlador, quien recibe los datos del formulario y los manda validar en el modelo con el método `autenticaUsuario()`.

Dependiendo del resultado de la autenticación se vuelve a desplegar el formulario de `login.jsp` con un mensaje de error o se crea una sesión de usuario y se redirecciona al menú principal.

Este ciclo de vida se repite por cada acción que el usuario realice sobre el sistema.

```
    public void doEdita(HttpServletRequest request, HttpServletResponse
response) {
      log.info("Ejecutando Edita Usuario");
      String uid=request.getParameter("uid");
      Integer idUsuario = new Integer(uid);
      Usuario usuario = getSeguridadService().
                            getBeanById(Usuario.class, idUsuario);
      Rol rol = getSeguridadService().
```

```
                        getBeanById(Rol.class, usuario.getIdRol());
      usuario.setRol(rol);

      List<Rol> roles = getSeguridadService().getAllBeans(Rol.class);

      //Paso de parametros al jsp
      setRequestAtribute("usuario", usuario);
      setRequestAtribute("roles", roles);
      setRequestAtribute("siguienteAccion", "update");
      setRequestAtribute("titulo", "Editando Usuario
"+usuario.getLogin());
      forward("seguridad/usuarioPage.jsp", request, response);
    }
```

Cuando un Controlador guarda un objeto bean en la petición por medio del método `setRequestAtribute()`, en la vista, el JSP puede declarar el uso de un bean, mediante la etiqueta `<jsp:useBean/>`, lo que permite obtener datos de él, para mostrarlos en donde se requieran:

```
<jsp:useBean id="usuario" class="com.neworks.model.seguridad.Usuario"
scope="request"/>

<form method="post" action="UsuariosControl" onSubmit="return
validaFormulario(this)">
      <input type="hidden" name="accion" value="<%=siguienteAccion%>" />
      <input type="hidden" name="id" value='<%=usuario.getId()%>' />
      <table class="formTable" summary=''>
            <tr>
                  <td>Nombre</td>
                  <td><input type="text" name="nombre"
value='<%=usuario.getNombre()%>' onChange="estaPresente(this)"/></td>
                  <td id="nombre_msg">*</td>
            </tr>
            <tr>
                  <td>Apellidos</td>
                  <td><input type="text" name="apellidos"
value='<%=usuario.getApellidos()%>' /></td>
            </tr>
...
```

Al enviar los datos del formulario de nuevo al Controlador, el Controlador puede usar el método `fillBeanFromRequest()` para recibirlos y guardarlos en un objeto que puede usar para continuar con el proceso:

```
    public void doUpdate(HttpServletRequest request,
                         HttpServletResponse response) {
        log.info("Ejecutando Update Usuario");
        Usuario usuario = new Usuario();
        fillBeanFromRequest(usuario, request);

        getSeguridadService().actualizaUsuario(usuario);
        redirect("UsuariosControl", response);
    }
```

En una aplicación empresarial es frecuente que una buena parte de la misma esté dedicada a administrar recursos de alguna índole (usuarios, equipos, clientes,

proveedores, productos etc.), lo que da lugar a pantallas de administración de catálogos, donde usualmente se pueden listar, dar de alta, editar y eliminar dichos recursos.

Para agilizar esta labor, el framework provee una utilería que genera el esqueleto básico de un catálogo dado el Modelo del objeto con que se requiere trabajar.

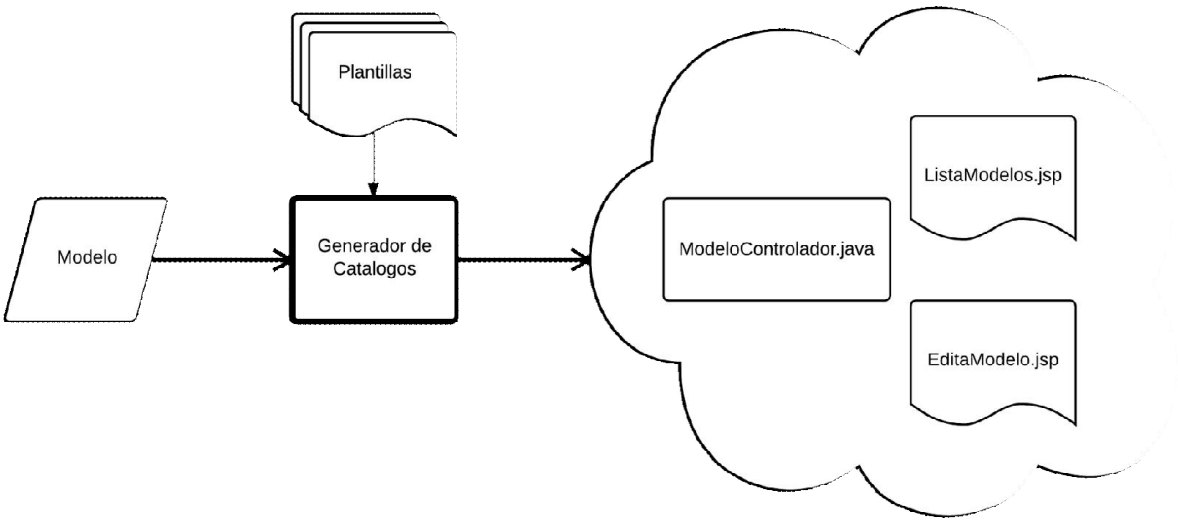

FIGURA V.11 - GENERADOR BÀSICO DE CATALOGOS

El generador básicamente lee el modelo para el que se requieren pantallas de administración y crea (a partir de las plantillas proporcionadas) un controlador y dos vistas, una para listar en una tabla los registros en de la base de datos y otra con un formulario para editar un registro (existente o nuevo).

Antes de generar las pantallas del catálogo, es posible modificar las plantillas para que adopten el estilo particular de la aplicación a la que serán agregadas.

Después de generado el catálogo, se debe personalizar cada pantalla para agregar las particularidades que los requerimientos indiquen.

VI. CONCLUSIONES

El diseño de un software preparado para el cambio y de fácil mantenimiento no es sencillo, requiere de mucha experiencia en el diseño, construcción y mantenimiento de sistemas, además de un estudio de las particularidades del software que se quiere construir (por ejemplo, el ambiente en el que se requiere usar, identificar y separar aquellas partes que tenderán a cambiar con el tiempo, la experiencia del equipo que lo construirá y de aquel que lo mantendrá, entre muchas otras variables).

Durante el presente proyecto de tesis se ha presentado a los frameworks como una herramienta fundamental en el desarrollo de sistemas, ya que permiten al programador que conoce y hace uso ellos, enfocarse en la lógica del negocio de la aplicación, permitiendo que los frameworks se encarguen de algunas decisiones de diseño y provean funcionalidad común a la mayoría de las aplicaciones (transaccionalidad, seguridad, arquitectura, etc.).

Cuando, en particular, el software a construir es un framework, que finalmente pretende facilitar la creación de alguna parte de una aplicación, es menester poner especial atención en el diseño del mismo, ya que para ser útil, debe ser adaptable a una definida variedad de situaciones y sobretodo, una vez en uso en diferentes proyectos de software, puede ser difícil modificar el framework sin afectar a las aplicaciones que lo usan.

Es por ello que se han expuesto las bases teóricas del diseño que resumen el conocimiento de los desarrolladores y teóricos de las últimas décadas, con lo que se cumple el principal objetivo de la presente tesis.

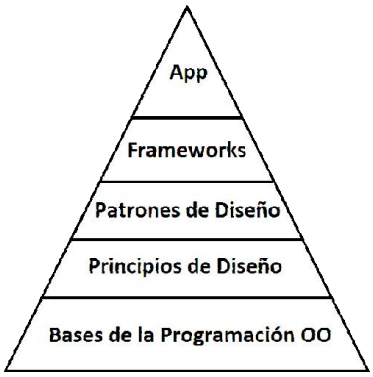

FIGURA VI.1 - APLICACIONES BASADAS EN LA EXPERIENCIA

Aquí en la punta superior de la pirámide se encuentra una aplicación basada en la acumulación de conocimiento teórico y práctico, que incluso un programador con poca experiencia puede ser capaz de aprovechar para desarrollar una aplicación con un excelente diseño arquitectónico si se monta sobre un framework que le proporcione dicha estructura.

Los frameworks, a su vez, usan patrones de diseño que les permiten ser extensibles a una variedad de usos en distintas aplicaciones de TI y les permiten variar sus implementaciones sin afectar su API público y así no afectar a quienes lo usan.

Los patrones de diseño, por su parte, proporcionan soluciones muy concretas a problemas de diseño que aparecen de forma recurrente, y están basados en la experiencia y principios de diseño.

Y finalmente, los principios de diseño proporcionan una pauta a seguir al realizar el diseño de un sistema. Al tenerlos en mente podemos tomar decisiones de diseño que nos ayudarán a construir un sistema más ordenado y con capacidades para crecer.

El caso de estudio planteado en el capítulo V permitió poner en práctica los conocimientos teóricos expuestos en esta tesis. Durante el diseño del framework propuesto fue una constante encontrarse con desafíos que facilitaran al programador el desarrollo de una aplicación web. En este capítulo se han recopilado varios de esos retos y cómo se les dio una solución y su relación con un patrón de diseño clásico.

En su conjunto, todas estas pequeñas soluciones, conforman la creación de un framework, que atiende los objetivos planteados originalmente:

- Arquitectura: Basándose totalmente en el patrón *Model 2*, define una arquitectura que separa las responsabilidades en Modelo, Vista y Controlador.

- Acceso a Datos: El acceso a datos se simplifica al proveer un manejador de conexiones que implementa el patrón *Singleton* para poder obtener una referencia al mismo desde cualquier parte del sistema y el patrón *Object Pool* que administra un grupo de conexiones a la base de datos.

  También añade una capa de abstracción que permite no realizar sentencias SQL directamente sobre la base de datos, sino que pretende definir las consultas, inserciones, actualizaciones mediante el uso de objetos, que es la forma natural en un lenguaje Orientado a Objetos, como lo es la plataforma Java.

  Esto se logró combinando varios patrones de diseño, que incluyen el *Template Method* para expresar condiciones SQL simples, el *Composite* que permite combinarlas para crear condiciones SQL compuestas, una fábrica de condiciones basada en el patrón *Abstract Factory* que permite cambiar fácilmente de proveedor de base de datos y una fábrica de beans, basada en el *método simple de fabricación* que permite construir objetos del modelo de datos dinámicamente para posteriormente llenarlos con datos obtenidos de la base.

- Transaccionalidad: La transaccionalidad en los servicios es simplificada mediante el uso del *Proxy* Dinámico proporcionado en el framework para tal fin, liberando al programador del código repetitivo que esto requiere y mejorando la legibilidad del código que contiene la lógica de negocio de la aplicación.

Para construir una aplicación con un lenguaje orientado a objetos es indispensable conocer las bases de la programación orientada a objetos, pero esto no es suficiente para diseñar sistemas robustos, de fácil mantenimiento y preparados para el cambio.

Se requiere además, de mucha experiencia que se puede adquirir con años de práctica y error o se puede recurrir a estudiar lo que otros desarrolladores han documentado y construir nuestra experiencia personal a partir de ese punto.

Aunque utilizar un buen framework es un gran punto de partida para un programador, ciertamente éste no resolverá todos los problemas de diseño. Más pronto que tarde, se presentarán problemas fuera del alcance del framework o se crearán nuevos problemas derivados de la utilización del mismo, por lo que resulta siempre útil, aunque no se diseñe un framework, estudiar poco a poco todas las capas del triángulo de conocimiento planteado, que representan la experiencia de muchos desarrolladores que a lo largo del tiempo sintetizan y documentan su conocimiento para transmitirlo a otros desarrolladores que tomarán estas ideas y las refinarán aún más para ir generando conocimiento nuevo y mejorar la forma en que construimos aplicaciones

VII. GLOSARIO

Programa (informático): Conjunto de instrucciones que una vez ejecutadas realizarán una o varias tareas en una computadora.

Software: Al conjunto general de programas, se le denomina software, que más genéricamente se refiere al equipamiento lógico o soporte lógico de una computadora digital.

Aplicación (informática): Un tipo de programa informático diseñado como herramienta para permitir a un usuario realizar uno o diversos tipos de trabajos muy específicos. Es una solución informática para la automatización de ciertas tareas complicadas como pueden ser la contabilidad, la redacción de documentos, o la gestión de un almacén.

Sistema (informático): Es el conjunto de componentes interrelacionados, que trabajando juntos, forman una aplicación informática.

Biblioteca: (del inglés library) es un conjunto de implementaciones funcionales, codificadas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se invoca.

Librería: Habitualmente se emplea el término librería para referirse a una biblioteca informática, por la similitud con el original inglés “library”. (Véase Biblioteca).

Código fuente: Es un conjunto de líneas de texto que son las instrucciones que debe seguir la computadora para ejecutar un programa informático (software), escritas en algún lenguaje de programación por un programador para describir por escrito su funcionamiento completo.

Lenguaje de programación: Es un lenguaje formal diseñado para expresar procesos que pueden ser llevados a cabo por máquinas como las computadoras.

API (del inglés Application Programming Interface): Es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

Arquitectura de Software: Es el diseño de más alto nivel de la estructura de un sistema. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como requerimientos no funcionales, como la confiabilidad, escalabilidad, portabilidad, y disponibilidad.

Programador: Es aquella persona que escribe, depura y mantiene el código fuente de un programa informático.

Desarrollador: Suele referirse a cualquier persona que pertenezca al equipo de desarrollo de un sistema, por lo que incluye no sólo programadores, si no también testers, líderes de proyecto, analistas, etc. Sin embargo en esta tesis se usa para referirse al programador que desarrolla un framework y diferenciarlo del programador de un sistema en particular (usuario del framework).

Usuario: Es el operador de una pieza de software, pudiendo ser una persona (a través de una GUI) o incluso otro sistema (a través de un API).

Thread-save: Una pieza de código es segura en cuanto a los hilos si funciona correctamente durante la ejecución simultánea de múltiples hilos que pueden acceder a datos compartidos.

Refactorización (del inglés Refactoring): Reestructurar el código fuente, alterando su estructura interna sin cambiar su comportamiento externo.

Bug: Un error o defecto en el software que hace que un programa se comporte incorrectamente.

Transacción: En un Sistema de Gestión de Bases de Datos (SGBD), es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible o atómica.

Transaccional: Un sistema se dice transaccional, si es capaz de mantener la integridad de los datos, creando grupos de operaciones (llamados Transacciones) que no pueden finalizar en un estado intermedio, o se ejecutan todas exitosamente o ninguna.

Código Usuario: Código de una aplicación, que hace uso de una librería o framework.

GUI: Interfaz Gráfica de Usuario, de inglés Graphic User Interface, es un programa informático que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Su principal uso, consiste en proporcionar un entorno visual sencillo para permitir la comunicación con el sistema operativo de una máquina o computador.

Bean: Los Java Beans son clases reutilizables que encapsulan muchos objetos en un solo objeto (el propio bean). Son serializables, tiene un constructor sin argumentos y permiten acceso a sus propiedades a través de métodos getter y setter.

VIII. BIBLIOGRAFÍA

- Apuntes sobre Computadoras y Programación (Tomo I)
  Solórzano, J. Fernando , Facultad de Ingeniería UNAM, 1995
- Software Engineering (9th Edition)
  Ian Sommerville (Mar 13, 2010)
- Practical API Design: Confessions of a Java Framework Architect
  Jaroslav Tulach, Apress; 1 edition (July 29, 2008)
- Building Application Frameworks: Object-Oriented Foundations of Framework Design
  Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson (1999)
- Agile Software Development: Principles, Patterns, and Practices
  Robert Martin (2002)
- Estudio comparativo de distintos frameworks para desarrollo web
  http://zeroturnaround.com/rebellabs/the-curious-coders-java-web-frameworks-comparison-spring-mvc-grails-vaadin-gwt-wicket-play-struts-and-jsf/
- Curvas de aprendizaje
  http://laiguanailustrada.blogspot.mx/2013/01/curvas-de-aprendizaje.html
- Design Principles
  http://www.oodesign.com/design-principles.html
- The Liskov Substitution Principle,
  Robert C. Martin, 1996
  http://www.objectmentor.com/resources/articles/lsp.pdf
- The Timeless Way of Building: A Pattern Language
  Christopher Alexander (Profesor de Arquitectura en Berkeley)
- Design Patterns: Elements of Reusable Object-Oriented Software
  Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, 1995
- Head First Design Patterns
  Eric Freeman, Elisabeth Freeman, 2004
- Using UML: Software Engineering with Object and Components
  Rob Pooley, Perdita Stevens, 1999
- Patrones de diseño en español (C++)
  http://arco.esi.uclm.es/~david.villa/pensar_en_C++/vol2/C10.html
- Design Patterns
  http://www.oodesign.com/
- Design Patterns Explained Simply
  http://sourcemaking.com/design_patterns
- Design Pattern Articles (.Net)
  http://www.blackwasp.co.uk/DesignPatternsArticles.aspx