ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA EN SISTEMAS

GUÍA PRÁCTICA PARA EL USO DE PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN SISTEMAS INFORMÁTICOS Y DE COMPUTACIÓN

ALDÁS MENA DANIEL ERNESTO
dany_1711@hotmail.com

ANDRADE CADENA MARITZA ALEJANDRA
alejitas_96@hotmail.com

DIRECTOR: Ing. MARCOS RAÚL CÓRDOVA BAYAS
raul.cordova@epn.edu.ec

Quito, FEBRERO 2010

DECLARACIÓN

Nosotros, Aldás Mena Daniel Ernesto, Andrade Cadena Maritza Alejandra, declaramos bajo juramento que el trabajo aquí descrito es de nuestra autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que hemos consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedemos nuestros derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

_______________________________ _______________________________

Daniel Aldás **Alejandra Andrade**

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Daniel Ernesto Aldás Mena y Maritza Alejandra Andrade Cadena, bajo mi supervisión.

Ing. Raúl Córdova Bayas
DIRECTOR DE PROYECTO

AGRADECIMIENTO

A mi familia por el ejemplo, las enseñanzas y los momentos compartidos conmigo, detalles que fueron la base de mi alegría y ganas de salir adelante cuando inicié esta etapa de mi vida, lejos de todo lo que había conocido, lejos de mi propia vida. Especialmente a Mama Clarita, Papi Tavo, Tío Lucho y Tío Oswaldo, ustedes fueron mis padres, cuando los propios estuvieron "indispuestos", mil gracias, lo que soy, en mucho se los debo a ustedes.

A mis padres, porque siempre confiaron en mi, porque al final del camino, volteo a ver las huellas dejadas por el recorrido y solo encuentro que ellos hicieron lo mejor que pudieron por mí.

A mis amigos, que gran parte de mi vida fueron mi única familia, las alegrías, las penas, los triunfos y fracasos de una vida, son mejores y más divertidos, si son con seres queridos, especialmente a los de la niñez, me conocen desde siempre, y siempre han estado ahí.

A mis compañeros de la U, compañeros de mil batallas, gracias por los triunfos, gracias por las derrotas, pero sobre todo, gracias por su amistad y aceptación.

A Alejandra, por mostrarme lo linda que puede ser una ciudad tan distinta a la que me preparó para la vida, gracias por todo lo compartido, esta tesis tiene gran valor por haberla hecho contigo.

A mis profes de la U, vaya que nos preparan para la vida, ¡Si que dieron pelea!, pero mil gracias.

Daniel Aldás

AGRADECIMIENTO

A Dios, porque su presencia en mi corazón me ha dado la fortaleza necesaria para seguir adelante superando los malos momentos y aprovechando al máximo las buenas experiencias.

A mi padre, por su esfuerzo y consejos que me mostraron que soy capaz de ponerme a la altura del obstáculo que se me presenta; su ejemplo de tenacidad y trabajo son mi principal motivación.

A mi madre, por su abnegación, sabiduría y compañía en cada momento de mi vida, su luz de amor me ha guiado por el camino del bien.

A mi hermana, por su cariño e incondicional apoyo, porque pese a ser tan diferentes siempre está para ayudarme.

A Daniel, por ayudarme a crecer sentimental, académica y profesionalmente, gracias por acompañarme en esta aventura y por estar presente en los momentos que han marcado mi vida.

A mi familia y amigos, por mostrarme lo maravillosa que es la vida cuando se cuenta con personas tan especiales que celebran mis triunfos y me tienden una mano para levantarme en mis tropiezos.

Alejandra Andrade

DEDICATORIA

A todos los que me conocen, no quiero poner nombres aquí, pues no acabaría nunca, pero dedico esta tesis a todos los que de una u otra manera estuvieron cerca de mi vida, los que me apoyaron siempre, los que no lo hicieron, a todos ustedes les debo lo que soy.

Daniel Aldás

DEDICATORIA

A mis padres y hermana, por acompañarme en este sueño que empezó unos años atrás, por estar conmigo cuando he llorado y he reído; pero sobretodo por disfrutar conmigo el dulce sabor que deja la satisfacción de una meta cumplida, por ustedes es mi compromiso de dejar una huella imborrable e intacta, ahora siento en mi corazón que cumplo la misión de hacer camino al andar de la mano de mi hermosa familia.

A Daniel, por todo lo que significas en mi vida, cada momento compartido es un recuerdo que atesoro, contar nuestra historia es revivir inigualables experiencias que se han grabado en mi mente y corazón.

Alejandra Andrade

CONTENIDO

RESUMEN 1
PRESENTACION 2
CAPÍTULO I 3
1. APLICACIÓN DE LOS PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE 3
1.1. USO DE PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE 3
1.1.1. ANTECEDENTES DE LOS PATRONES DE DISEÑO 4
1.1.2. DEFINICIÓN PATRÓN DE DISEÑO 5
1.1.3. ELEMENTOS DE LOS PATRONES DE DISEÑO 6
a) Nombre del patrón 7
b) Función 7
c) Estructura 7
1.1.4. CLASIFICACIÓN PATRONES DE DISEÑO 7
1.1.4.1. Patrones de Creación 7
a) Abstract Factory 7
b) Builder 8
c) Factory Method 9
d) Prototype 9
e) Singleton 10
1.1.4.2. Patrones Estructurales 10
a) Adapter 10
b) Bridge 11
c) Composite 11
d) Decorator 12
e) Facade 12
f) Flyweight 13
g) Proxy 13
1.1.4.3. Patrones de Comportamiento 14
a) Chain of Responsibility 14
b) Command 15
c) Interpreter 15
d) Iterator 16
e) Mediator 17
f) Memento 17
g) Observer 17
h) State 18
i) Strategy 18
j) Template Method 19
k) Visitor 20
1.2. BENEFICIOS DEL USO DE PATRONES DE DISEÑO 20
a) PERMITEN MANTENER UNA ALTA COHESIÓN 21
b) FACILITAN LA REUTILIZACIÓN DE SOFTWARE 21
c) FACILITAN LA IDENTIFICACIÓN DE OBJETOS 22
d) PERMITEN DISTRIBUIR RESPONSABILIDADES 22

e) GARANTIZAN REUSABILIDAD, EXTENSIBILIDAD Y MANTENIMIENTO ... 22
f) PROVEEN SOLUCIONES CONCRETAS ... 22
g) OFRECEN MAYOR UTILIDAD QUE LOS LENGUAJES DE PROGRAMACIÓN ... 23
h) PROVEEN UNA SOLUCIÓN MÁS EQUILIBRADA ... 23
i) FACILITAN EL ENTENDIMIENTO DE LA SOLUCIÓN GRACIAS AL ENCAPSULAMIENTO ... 23
1.3. DESARROLLO DE SOFTWARE CON PATRONES DE DISEÑO VS DESARROLLO SIN PATRONES ... 23
1.3.1. DESARROLLO CON PATRONES ... 23
1.3.1.1. Ventajas ... 24
a) Capturar las mejores prácticas ... 24
b) Soluciones probadas ... 24
c) Fácil integración ... 24
d) Incremento de la eficiencia ... 25
e) Proceso de diseño serio ... 25
1.3.1.2. Desventajas ... 25
a) Incremento de la complejidad debido a la falta de experiencia ... 25
b) Mapeo e integración de soluciones ... 25
c) Disminución de la capacidad de iniciativa ... 26
1.3.2. DESARROLLO SIN PATRONES ... 26
1.3.2.1. Ventajas ... 26
a) Falta de experiencia ... 26
b) Reducción del riesgo ... 26
1.3.2.2. Desventajas ... 27
a) Uso de estándares ... 27
b) Mantenimiento ... 27
1.3.3. COMPARACIÓN ... 27
CAPÍTULO II ... 29
2. ESTRUCTURACIÓN DE LA GUÍA ... 29
2.1. CARACTERÍSTICAS Y FUNCIONES DE LAS GUÍAS DE REFERENCIA ... 29
2.1.1. CARACTERÍSTICAS ... 29
2.1.2. FUNCIONES ... 30
2.2. DISEÑO DE GUÍAS DE REFERENCIA ... 30
a) FONDO: ... 31
b) FORMA: ... 31
c) INTRODUCCIÓN ... 31
d) ALCANCE ... 31
e) CONTENIDO ... 32
2.3. REQUISITOS PARA LA ELABORACIÓN DE GUÍAS DE REFERENCIA. ... 32
2.4. FACTORES A CONSIDERARSE PARA LA ELABORACIÓN DE LA GUÍA PARA EL USO DE PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE ... 33
CAPÍTULO III ... 35
3. GUÍA PARA EL USO DE PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE ... 35
3.1. OBJETIVOS ... 35
3.2. ALCANCE ... 35

3.3. DEFINICIONES 36
3.4. CONTENIDO 37
I INTRODUCCIÓN 37
II GUÍA PARA EL USO DE PATRONES 38
1. Catálogo de Patrones 38
1.1. Patrones de Creación 39
1.1.1. Abstract Factory 39
1.1.2. Builder 41
1.1.3. Factory Method 43
1.1.4. Prototype 44
1.1.5. Singleton 46
1.2. Patrones Estructurales 47
1.2.1. Adapter 47
1.2.2. Bridge 49
1.2.3. Composite 51
1.2.4. Decorator o Wrapper 53
1.2.5. Facade 54
1.2.6. Flyweight 56
1.2.7. Proxy O Sustituto 58
1.3. Patrones de Comportamiento 60
1.3.1. Chain Of Responsability 60
1.3.2. Command 62
1.3.3. Interpreter 63
1.3.4. Iterator 65
1.3.5. Mediator 67
1.3.6. Memento 68
1.3.7. Observer O Dependientes 70
1.3.8. State 71
1.3.9. Strategy 73
1.3.10. Template 75
1.3.11. Visitor 76
2. Caracterización del sistema 78
2.1. Objetivo del sistema 78
2.2. Requerimientos del sistema 79
2.2.1. Requerimientos Funcionales 80
2.2.2. Requerimientos no Funcionales 80
3. Modelado de la solución 81
3.1. Creación de los casos de uso 81
4. Análisis detallado 83
4.1. Identificación de funcionalidades 83
4.1.1. Identificación de componentes 83
4.2. Identificación de problemas 84
4.2.1. Selección de patrones candidatos 85
4.2.2. Análisis del catálogo de patrones de diseño 86
4.3. Diseño del diagrama de clases de la aplicación 86
4.3.1. Elaborar el diagrama de clases 89

4.3.2. Definición de patrones participantes 90
5. Arquitectura de la solución 90
5.1. Definición de las capas de la aplicación 90
5.2. Elaboración del diagrama de despliegue 92
6. Diseño de la solución 93
6.1. Diseño del modelo Entidad-Relación 93
6.2. Diseño del modelo conceptual 94
6.2.1. Definición de las tablas 94
1.1.1. Elaboración del modelo 94
6.3. Diseño del modelo físico 95
7. Codificación del sistema 95
7.1. Creación de la base de datos de la aplicación 95
7.2. Implementación de las capas definidas en la arquitectura de la solución 95
CAPÍTULO IV 97
4. APLICACIÓN DE LA GUÍA EN UN CASO DE ESTUDIO 97
4.1. DESCRIPCIÓN DEL CASO DE ESTUDIO 97
4.1.1. IDENTIFICACIÓN DEL CASO DE ESTUDIO 97
4.2. APLICACIÓN DE LA GUÍA EN LA IMPLEMENTACIÓN DEL CASO DE ESTUDIO 98
4.2.1. CARACTERIZACIÓN DEL SISTEMA 98
4.2.1.1. Objetivo 98
4.2.1.2. Requerimientos del Sistema 98
4.2.1.2.1. Requerimientos Funcionales 99
4.2.1.2.1.1. Responsabilidades 99
4.2.1.2.1.2. Exclusiones 100
4.2.1.2.2. No Funcionales 100
4.2.2. MODELADO DE LA SOLUCIÓN 101
4.2.2.1. CASOS DE USO 101
4.2.3. ANÁLISIS DETALLADO 103
4.2.3.1. FUNCIONALIDADES 103
4.2.3.1.1. Identificación de componentes 104
4.2.3.2. IDENTIFICACIÓN DE PROBLEMAS 104
4.2.3.2.1. Selección de patrones candidatos 105
4.2.3.2.2. Análisis del catálogo de patrones 105
4.2.3.3. DIAGRAMA DE CLASES DE LA APLICACIÓN 105
4.2.3.3.1. Diagrama del Módulo de seguridades 105
4.2.3.3.2. Definición de Patrones Participantes 106
4.2.4. ARQUITECTURA DE LA SOLUCIÓN 107
4.2.4.1. DEFINICIÓN DE CAPAS DE LA APLICACIÓN 107
4.2.4.2. DIAGRAMA DE DESPLIEGUE 107
4.2.5. DISEÑO DE LA SOLUCIÓN 108
4.2.5.1. MODELO CONCEPTUAL (REQUERIMIENTO DE CONTROL DE ACCESO POR PERFILES) 108
4.2.5.2. MODELO FÍSICO (REQUERIMIENTO DE CONTROL DE ACCESO POR PERFILES) 109
4.2.6. CODIFICACIÓN DE SISTEMA 110

4.2.6.1. CREACIÓN DE LA BDD DEL REQUERIMIENTO 110
4.2.6.2. CODIFICACIÓN DEL REQUERIMIENTO DE CONTROL DE ACCESO POR PERFILES 112
4.3. ANÁLISIS DEL USO DE LA GUÍA 136
CAPÍTULO V 138
5. CONCLUSIONES Y RECOMENDACIONES 138
5.1. CONCLUSIONES 138
5.2. RECOMENDACIONES 139
BIBLIOGRAFÍA 140
TESIS: 140
LIBROS: 140
ARTÍCULOS: 140
PÁGINAS WEB: 140
GLOSARIO 143
ANEXO 1: CASO DE ESTUDIO 144

CONTENIDO DE FIGURAS

CAPITULO I

Figura 1.1: Diagrama UML Patrón Singleton 5
Figura 1.2: Codificación Patrón Singleton 6
Figura 1.3: Estructura Patrón Abstract Factory 8
Figura 1.4: Builder 8
Figura 1.5: Estructura Patrón Factory Method 9
Figura 1.6: Estructura Patrón Prototype 9
Figura 1.7: Estructura Patrón Singleton 10
Figura 1.8: Estructura Patrón Adapter 11
Figura 1.9: Estructura Patrón Bridge 11
Figura 1.10: Estructura Patrón Composite 12
Figura 1.11: Estructura Patrón Decorator 12
Figura 1.12: Estructura Patrón Facade 13
Figura 1.13: Estructura Patrón Flyweight 13
Figura 1.14: Estructura Patrón Proxy 14
Figura 1.15: Estructura Patrón Chain of Responsability 15
Figura 1.16: Estructura Patrón Command 15
Figura 1.17: Estructura Patrón Interpreter 16
Figura 1.18: Estructura Patrón Iterator 16
Figura 1.19: Estructura Patrón Mediator 17
Figura 1.20: Estructura Patrón Memento 17
Figura 1.21: Estructura Patrón Observer 18
Figura 1.22: Estructura Patrón State 18
Figura 1.23: Estructura Patrón Strategy 19
Figura 1.24: Estructura Patrón Template Method 19
Figura 1.25: Estructura Patrón Visitor 20

CAPITULO III

Figura 3.1: Patrón Abstract Factory 40
Figura 3.2: Patrón Builder 42
Figura 3.3: Patrón Factory Method 43
Figura 3.4: Patrón Prototype 45
Figura 3.5: Patrón Singleton 46
Figura 3.6: Patrón Adapter 48
Figura 3.7: Patrón Bridge 50
Figura 3.8: Patrón Composite 52
Figura 3.9: Patrón Decorator 53
Figura 3.10: Patrón Facade 55
Figura 3.11: Patrón Flyweight 57
Figura 3.12: Patrón Proxy 59
Figura 3.13: Patrón Chain of Responsability 61

Figura 3.14: Patrón Command ... 62
Figura 3.15: Patrón Interpreter ... 64
Figura 3.16: Patrón Iterator ... 66
Figura 3.17: Patrón Mediator ... 67
Figura 3.18: Patrón Memento ... 69
Figura 3.19: Patrón Observer ... 70
Figura 3.20: Patrón State ... 72
Figura 3.21: Patrón Strategy ... 74
Figura 3.22: Patrón Template ... 75
Figura 3.23: Patrón Visitor ... 77
Figura 3.24: Ejemplo Diagrama de Casos de Uso ... 82
Figura 3.25: Ejemplo Diagrama de Clases ... 90
Figura 3.26: Capa de Presentación ... 91
Figura 3.27: Capa de Negocio ... 91
Figura 3.28: Patrón Builder ... 91
Figura 3.29: Diagrama de Despliegue ... 93
Figura 3.30: Modelo Entidad-Relación ... 93
Figura 3.31: Modelo Conceptual ... 94
Figura 3.32: Modelo Físico ... 95
Figura 3.33: Codificación patrón Abstract Factory ... 96

CAPITULO IV

Figura 4.1: Diagrama de caso de uso para el requerimiento funcional 1. ... 101
Figura 4.2: Diagrama de caso de uso para el requerimiento funcional 2 ... 102
Figura 4.3: Diagrama de caso de uso para el requerimiento funcional 3 ... 102
Figura 4.4: Implementación del requerimiento de gestión de permisos de perfiles de usuario ... 106
Figura 4.5: Diagrama de Arquitectura de la solución ... 107
Figura 4.6: Diagrama de despliegue de la aplicación ... 108
Figura 4.7: Modelo conceptual de la solución ... 109
Figura 4.8: Modelo físico de la solución ... 110
Figura 4.9: Tablas ejemplo del modelo de BDD ... 111

CONTENIDO DE TABLAS

CAPITULO I

Tabla 1.1: Tabla Comparativa, Desarrollo de software con patrones y desarrollo sin patrones ... 28

CAPITULO III

Tabla 3.1: Descripción Actores y Casos de uso ... 81
Tabla 3.2: Descripción de relaciones entre Casos de uso ... 82
Tabla 3.3: Descripción Clases ... 87

Tabla 3.4: Clasificación de las soluciones presentadas por los patrones de diseño 88
Tabla 3.5: Descripción relaciones entre clases 89
Tabla 3.6: Ejemplo de una tabla 94

CONTENIDO DE ILUSTRACIONES

CAPITULO IV

Ilustración 4.1: Código de la clase PerfilBdd, capa de acceso a datos para el objeto “Perfil” 115
Ilustración 4.2: Código de la clase PermisoBdd, capa de acceso a datos para el objeto “Permiso” 121
Ilustración 4.3: Código de la clase CategoríaPermisoBdd, capa de acceso a datos para el objeto “CategoriaPermiso” 124
Ilustración 4.4: Código de la clase PermisoPerfilBdd, capa de acceso a datos para el objeto “PermisoPerfil” 126
Ilustración 4.5: Código de la clase UsuarioBdd, capa de acceso a datos para el objeto “Usuario” 133
Ilustración 4.6: Código de la Patrón CMMPerfil, de tipo Command 134
Ilustración 4.7: Código de la Patrón CMMPermiso, de tipo Command 134
Ilustración 4.8: Definición de la clase abstracta “Permiso” (Patrón de tipo Composite) 134
Ilustración 4.9: Implementación del Patrón composite, para un objeto simple 135
Ilustración 4.10: Implementación del Patrón composite, para un objeto compuesto 136

RESUMEN

En la constante lucha para mejorar la calidad del software, se han desarrollado varias estrategias, muchas de las cuales son producto de mejoras realizadas sobre estrategias anteriores; una de estas, y probablemente la más versátil son los patrones de diseño, que consisten en la reutilización de soluciones, no solamente de código fuente, sino, soluciones integrales a problemas repetitivos en el desarrollo de software. Los patrones han sido la base para la creación de librerías reutilizadas que agilitan el versionamiento de aplicaciones. Así tenemos por ejemplo a Microsoft que introduce en el mercado “Enterprise Library”, la cual constituye en una compilación de patrones y prácticas aplicables al desarrollo de software; por otro lado, Java nos presenta una infinidad de “plug-in’s” o complementos a sus plataformas JVM, que son una serie de patrones de programación, muchos incluso desarrollados por el propio Erich Gamma.

Sin embargo, tanto Microsoft como Java, han desarrollado de una u otra manera sus propios patrones de diseño; los cuales, independiente de los beneficios que ofrezcan al momento de aplicarlos, se convierten en elementos de caja negra, muchas veces incomprensibles para sus usuarios. Este proyecto de titulación ha desarrollado una guía que servirá como herramienta para lograr un fácil entendimiento de los patrones de diseño, su origen, definición y aplicación.

Esta guía está orientada a personas interesadas en entender y/o aplicar patrones de diseño en el desarrollo de software.

Es importante indicar que el desarrollo del presente proyecto se basa en el estudio de los patrones de diseño originales descritos en el libro “Design Patterns” escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides. Para la aplicación de la guía a un caso de estudio se utilizó UML, Java y MySql.

PRESENTACION

El presente proyecto de titulación busca agilitar el entendimiento de patrones de diseño, por parte de los desarrolladores que los utilizan, mediante una guía práctica para la correcta utilización de patrones en el desarrollo de software, además, este proyecto presenta un listado de los patrones de diseño originales, indicando su finalidad, su forma de uso, su integración o colaboración con otros patrones.

La estructura del presente trabajo podría resumirse de la siguiente manera:

Capítulo I: Aplicación de los Patrones de Diseño en el Desarrollo de Software; en este capítulo se presenta una breve reseña histórica del origen de los patrones de diseño, además de un análisis de las ventajas y desventajas del uso de los mismos.

Capítulo II: Estructuración de la guía; este capítulo presenta las bases a emplearse en la creación de la guía.

Capítulo III: Guía para el uso de los patrones de diseño en el desarrollo de Software; este capítulo se compone de la guía, la cual, a su vez, está compuesta por un catálogo de patrones de diseño.

Capítulo IV: Aplicación de la guía en un caso de estudio; en este capítulo se presenta un caso de estudio para utilizar la guía, el resultado de la aplicación de la guía, y un análisis del uso de la misma, para poder desarrollar conclusiones y recomendaciones para posteriores usuarios de la guía.

Capítulo V: Conclusiones y recomendaciones; las conclusiones y recomendaciones presentadas tras el desarrollo del presente proyecto de titulación.

CAPÍTULO I

1. APLICACIÓN DE LOS PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE

El presente capítulo contiene una descripción detallada de los patrones de diseño de Software, su uso y sus beneficios; también incluye un análisis comparativo entre el desarrollo de software con y sin patrones, con el objetivo de obtener las ventajas y desventajas de cada técnica, para conocer el verdadero aporte de los patrones de diseño.

1.1. USO DE PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE

Al igual que en casi toda actividad de ingeniería, en el área de software y, especialmente, en el desarrollo se presentan situaciones similares, problemas e inconvenientes repetitivos que, de una u otra forma demandan la búsqueda de una solución.

Aún cuando actualmente los desarrolladores disponen de poderosas herramientas que facilitan el desarrollo de un sistema, manteniendo altos niveles de calidad, no simplifican el proceso de codificación de software.

En este contexto se está trabajando actualmente bajo el término “Patrones”, los cuales constituyen herramientas de soporte en el desarrollo de software orientado a objetos, y su principio es muy básico y comprensible, se los puede llamar “Soluciones reutilizables”.

1.1.1. ANTECEDENTES DE LOS PATRONES DE DISEÑO

El inicio de los patrones de diseño se dio en 1979, cuando el arquitecto Christopher Alexander publicó el libro “The Timeless Way of Building”, que se constituyó en un gran aporte para la arquitectura; ya que proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad; en el que se asegura que "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizarla un millón de veces más sin tener que pensarla otra vez."[1]

Los patrones que Christopher Alexander y sus colegas definieron, publicados en un volumen denominado "A Pattern Language", son un intento de formalizar y plasmar de manera práctica generaciones de conocimiento arquitectónico.

Tras el éxito de los patrones en la arquitectura, en 1987, Ward Cunningham y Kent Beck se basaron en varias ideas de Alexander para desarrollar cinco patrones de interacción hombre-computador (HCI) y publicaron un artículo titulado “Using Pattern Languages for OO Programs[2]”.

Sin embargo, no fue hasta principios de los 90's cuando los patrones de diseño tuvieron un gran éxito en el mundo de la informática tras la publicación del libro “Design Patterns” escrito por el grupo denominado GoF[3], que en español significa “La pandilla de los cuatro”, estaba conformado por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones diseño comunes para el desarrollo de software.

---

[1] **Christopher Alexander, “The Timeless Way of Building”**

[2] **OOPSLA-87, artículo “Using Patterns Languages for OO Programs”**

[3] **GoF: Gang of Four**

1.1.2. DEFINICIÓN PATRÓN DE DISEÑO

Los patrones de diseño consisten en la descripción de un problema de diseño dentro de un contexto dado y la definición de una posible solución correcta al mismo.

Los patrones no son principios abstractos que requieran su redescubrimiento para obtener una aplicación satisfactoria, ni son específicos a una situación particular o cultural; son algo intermedio.

A pesar de los beneficios de los patrones, el simple hecho de usarlos, no garantiza el éxito en el desarrollo de una aplicación orientada a objetos, peor aún, el usar patrones sin fundamentos puede llegar a convertirse en un problema superior a lo que sería diseñar una solución desde cero.

El *Ejemplo 1.1*[4], Patrón de instancia única (Singleton); resulta útil en las aplicaciones que necesitan acceder fácilmente a un objeto único, es decir la instancia única de cierta clase, como por ejemplo: un gestor de impresoras, un pool de conexiones a una base de datos, un gestor de parámetros de configuración, etc.

Estructura:

La representación de un patrón de instancia única mediante un diagrama UML se muestra en la Figura 1.1.

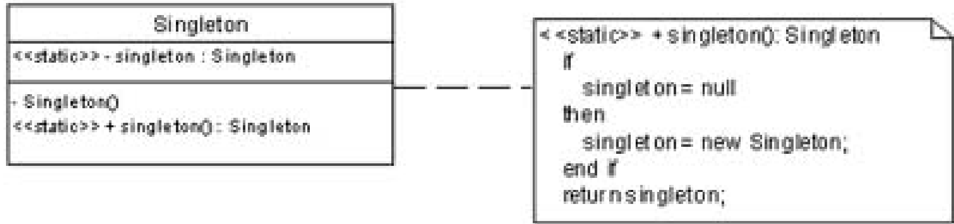


Figura 1.1: Diagrama UML Patrón Singleton

---

[4] **Ejemplo 1: http://www.elrincondelprogramador.com/default.asp?id=45&pag=articulos%2Fleer.asp**

Podemos observar dos características básicas:

- *Restricción de acceso al constructor:* Con esto conseguimos que sea imposible crear nuevas instancias del objeto, pues solo la propia clase puede crear la instancia.
- *Mecanismo de acceso a la instancia:* El acceso a la instancia única se hace a través de un único punto bien definido, que es gestionado por la propia clase y que puede ser accedido desde cualquier parte del código.

Código:

La representación de un patrón de instancia única mediante código se encuentra ilustrada en la figura 1.2.

```
public class Singleton {

    static private Singleton singleton = null;

    private Singleton() { }

    static public Singleton getSingleton() {

        if (singleton == null) {
              singleton = new Singleton();
        }
        return singleton;
    }

    /*
     * Métodos del singleton.
     */

    public String metodo() {
        return "Singleton instanciado bajo demanda";
    }
 }
```

Figura 1.2: Codificación Patrón Singleton

1.1.3. ELEMENTOS DE LOS PATRONES DE DISEÑO

Un patrón de diseño generalmente se encuentra conformado por los siguientes puntos esenciales para obtener una clara descripción:

a) Nombre del patrón

Describe, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias.

b) Función

Detalla cuándo aplicar el patrón, es una explicación del problema y su contexto.

c) Estructura

Describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones.

1.1.4. CLASIFICACIÓN PATRONES DE DISEÑO

Para un mejor entendimiento, los patrones de diseño se clasifican en tres grandes grupos.

1.1.4.1. Patrones de Creación

Los patrones de creación están asociados al proceso de creación de los objetos, entre objetos se delegan los procesos de creación. Entre los principales patrones que se encuentran en este grupo podemos listar los siguientes:

a) Abstract Factory

Nombre: Fabricación Abstracta
Función: Trabajar con objetos de distintas familias de manera que no se mezclen entre sí, haciendo transparente el tipo de familia concreta que se esté usando.
Estructura: La estructura que cumple este patrón se muestra en la Figura 1.3.

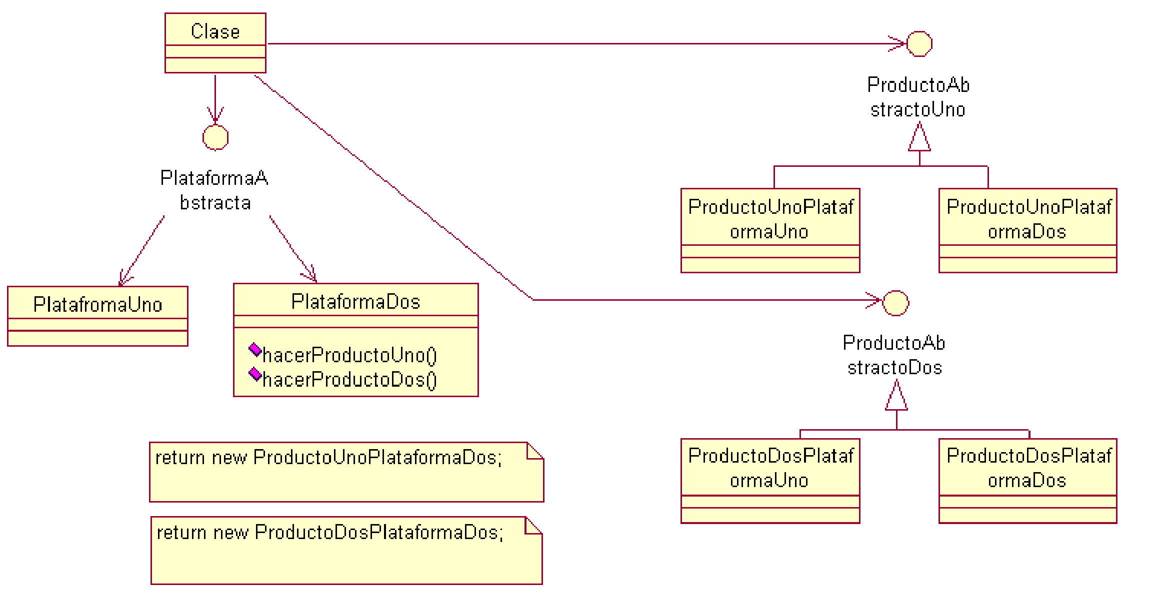


Figura 1.3: Estructura Patrón Abstract Factory

b) *Builder*

Nombre: Constructor virtual

Función: Facilitar la abstracción del proceso de creación de un objeto complejo, centralizándolo en un único punto.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.4.

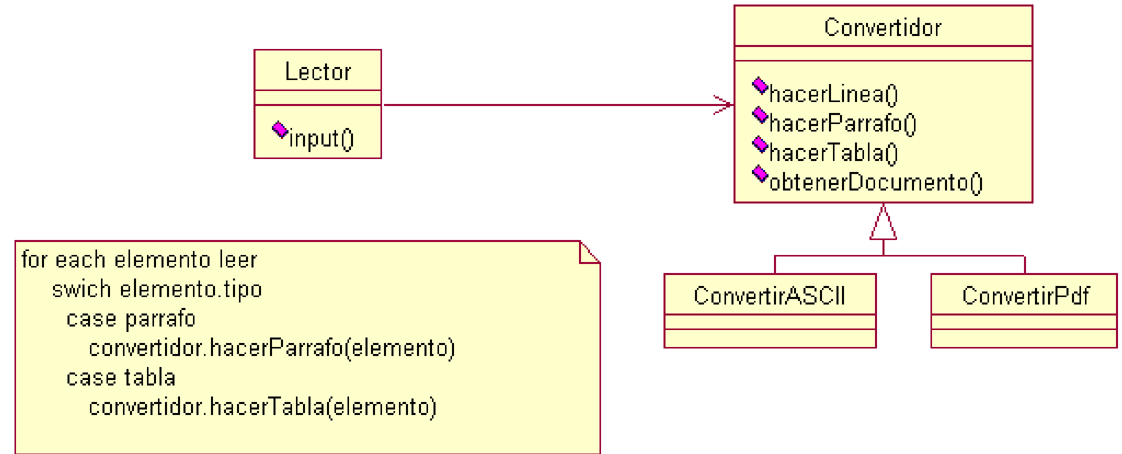


Figura 1.4: Builder

c) Factory Method

Nombre: Método de fabricación

Función: Centralizar en una clase constructora la creación de objetos de un subtipo de un tipo determinado.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.5.

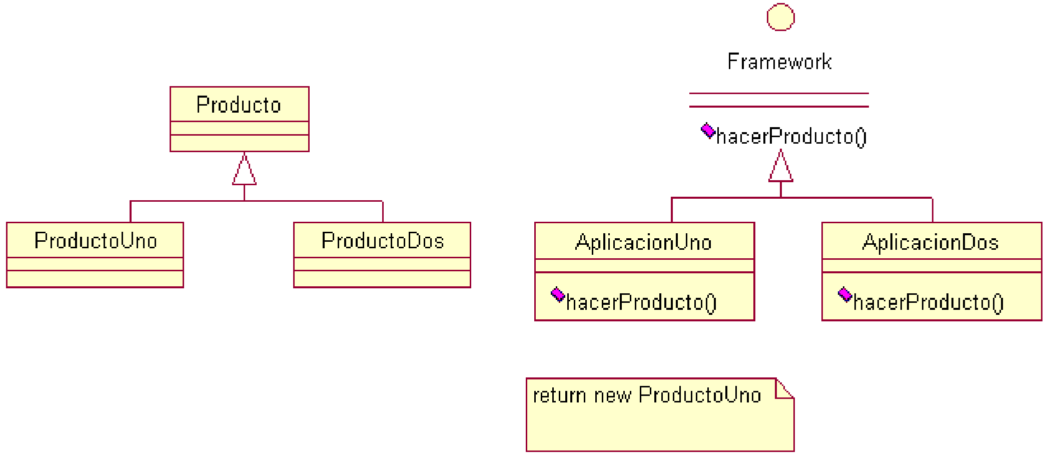


Figura 1.5: Estructura Patrón Factory Method

d) Prototype

Nombre: Prototipo

Función: Crear nuevos objetos clonándolos de una instancia ya existente.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.6.

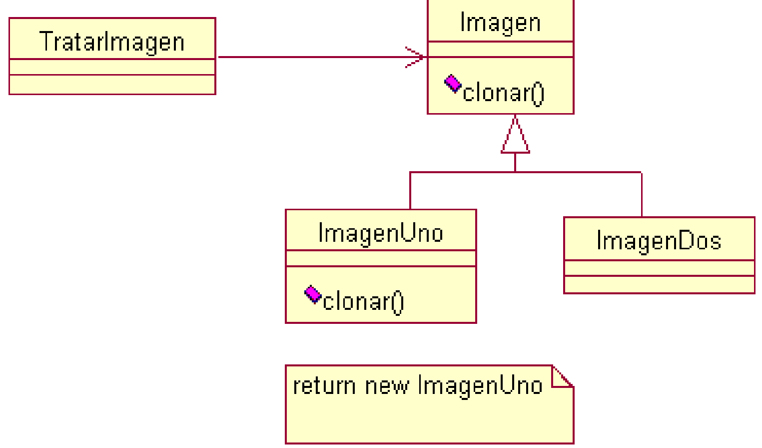


Figura 1.6: Estructura Patrón Prototype

e) Singleton

Nombre: Instancia única

Función: Garantizar la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a la misma.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.7.

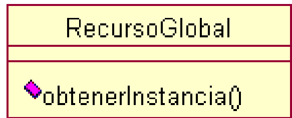


Figura 1.7: Estructura Patrón Singleton

1.1.4.2. Patrones Estructurales

Los patrones estructurales tratan la composición de clases u objetos; para lo cual hacen uso de dos recursos, dependiendo del tipo de composición, es decir que para clases es necesaria la herencia mientras que para los objetos se describen formas de ensamblar objetos.

Entre los principales patrones que se encuentran en este grupo podemos listar los siguientes:

a) Adapter

Nombre: Adaptador

Función: Adaptar una interface para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.8.

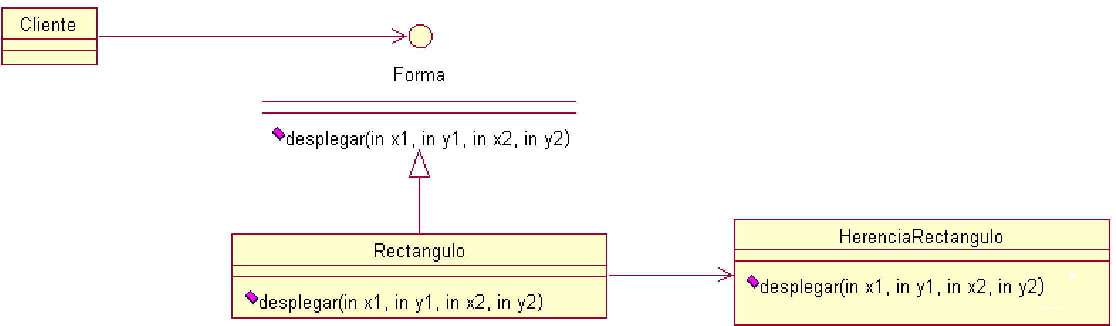


Figura 1.8: Estructura Patrón Adapter

b) Bridge

Nombre: Puente

Función: Desacoplar una abstracción de su implementación.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.9.

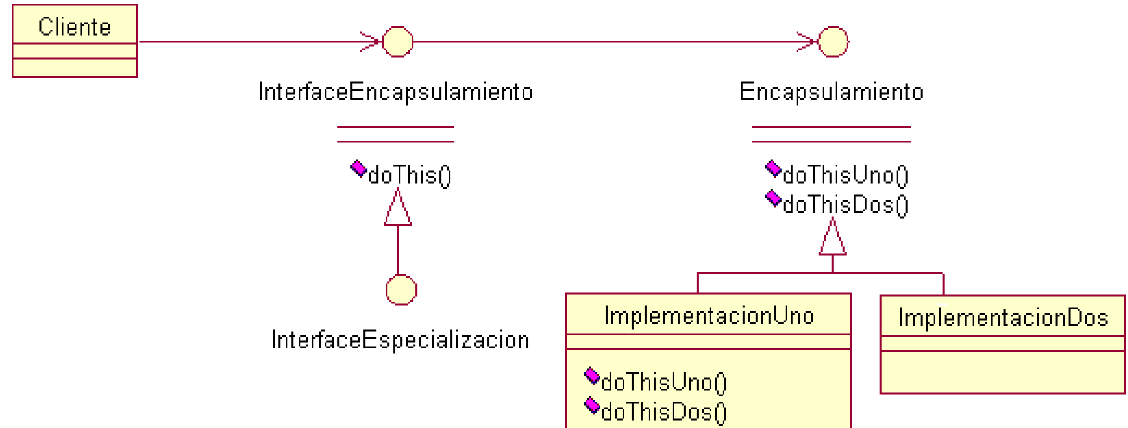


Figura 1.9: Estructura Patrón Bridge

c) Composite

Nombre: Objeto compuesto

Función: Manejar objetos compuestos como si se tratase de un simple.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.10.

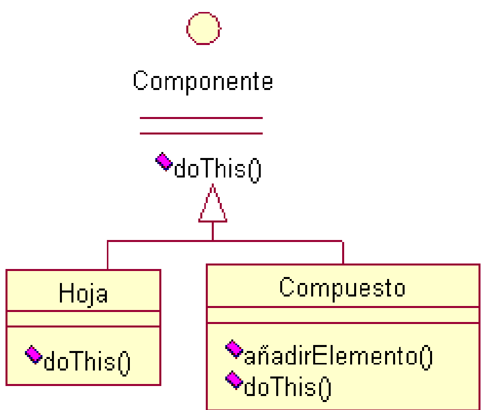


Figura 1.10: Estructura Patrón Composite

d) Decorator

Nombre: Envoltorio

Función: Facilitar la añadidura de funcionalidad a una clase dinámicamente.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.11.

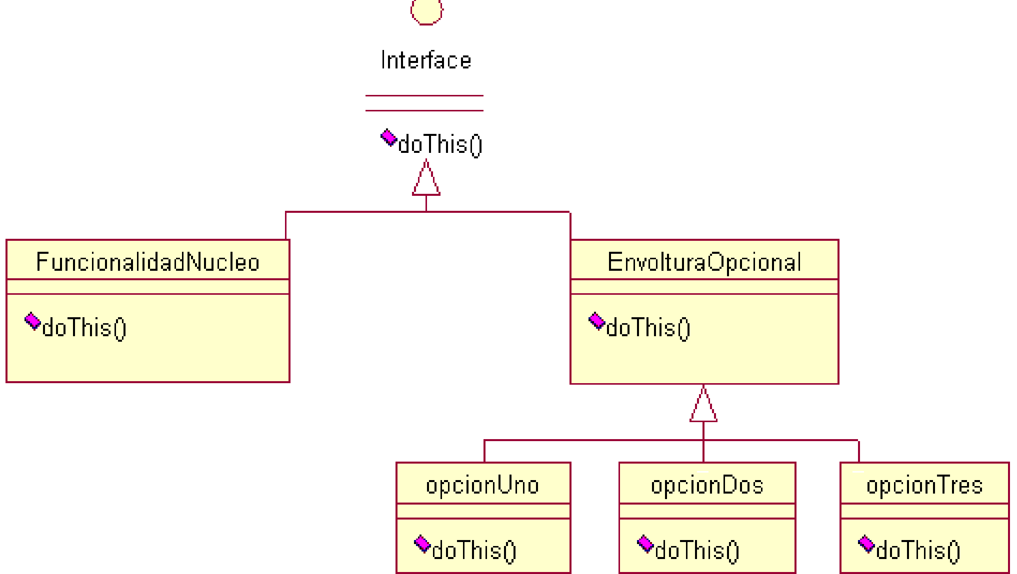


Figura 1.11: Estructura Patrón Decorator

e) Facade

Nombre: Fachada

Función: Proveer de una interface unificada y simple para acceder a una interface o grupo de interfaces de un subsistema.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.12.

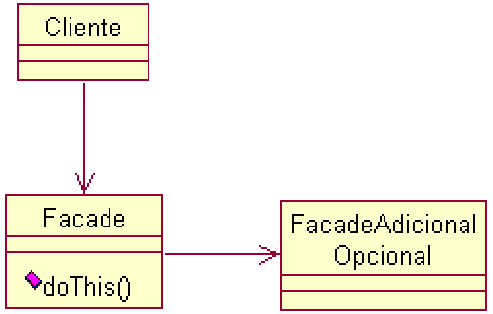


Figura 1.12: Estructura Patrón Facade

f) Flyweight

Nombre: Peso ligero

Función: Reducir la redundancia en situaciones en las que se presentan grandes cantidades de objetos que poseen idéntica información.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.13.

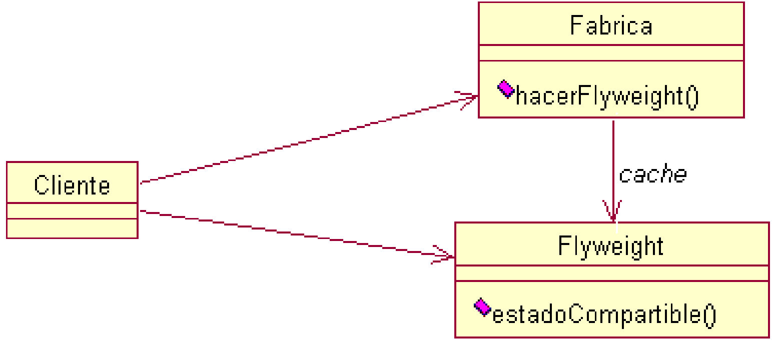


Figura 1.13: Estructura Patrón Flyweight

g) Proxy

Nombre: Proxy

Función: Mantiene un representante de un objeto.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.14.

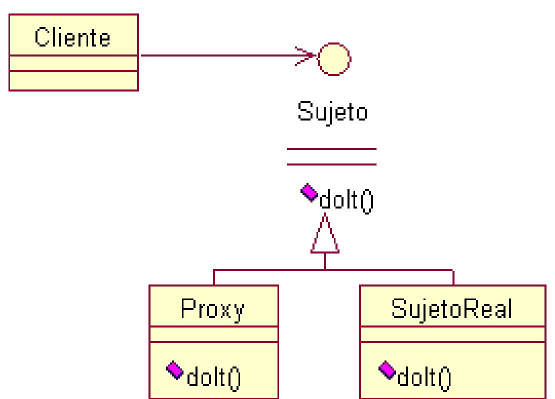


Figura 1.14: Estructura Patrón Proxy

1.1.4.3. Patrones de Comportamiento

Los patrones comportamiento caracterizan el modo en que las clases y objetos interactúan y se distribuyen la responsabilidad. Este tipo de patrones, en las clases hacen uso de la herencia para describir algoritmos y flujos de control; mientras que los de objetos describen cómo cooperan un grupo de objetos para realizar una tarea que ninguno podría llevar a cabo por sí solo.

Entre los principales patrones que se encuentran en este grupo podemos listar los siguientes:

a) Chain of Responsibility

Nombre: Cadena de responsabilidad

Función: Establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.15.

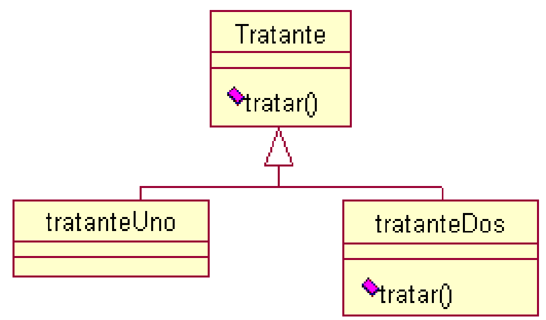


Figura 1.15: Estructura Patrón Chain of Responsability

b) Command

Nombre: Orden

Función: Encapsular una operación en un objeto, permitiendo ejecutarla sin necesidad de conocer el contenido de la misma.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.16.

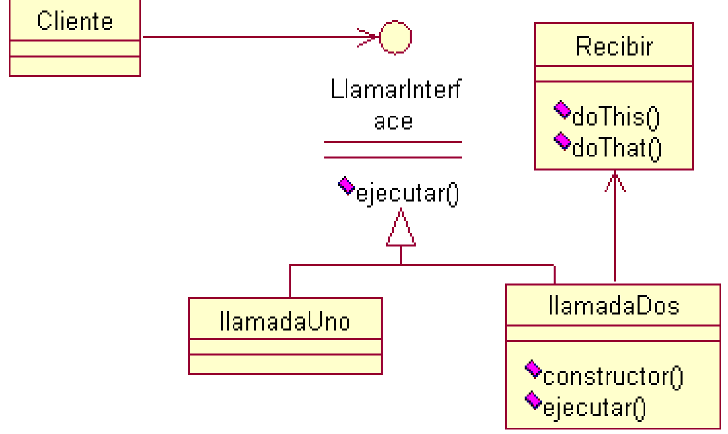


Figura 1.16: Estructura Patrón Command

c) Interpreter

Nombre: Intérprete

Función: Definir una gramática para un lenguaje dado, así como las herramientas necesarias para interpretarlo.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.17.

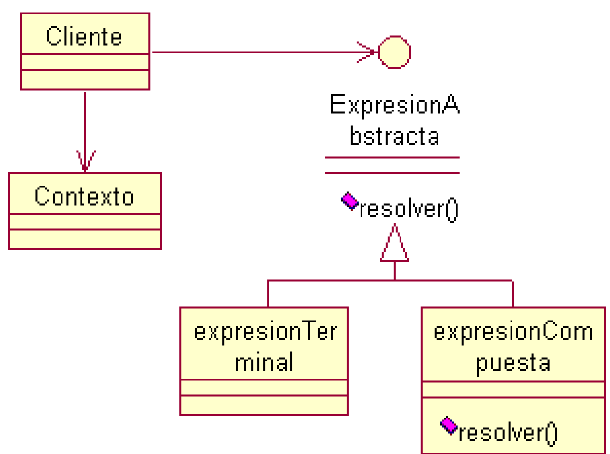


Figura 1.17: Estructura Patrón Interpreter

d) Iterator

Nombre: Iterador

Función: Realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.18.

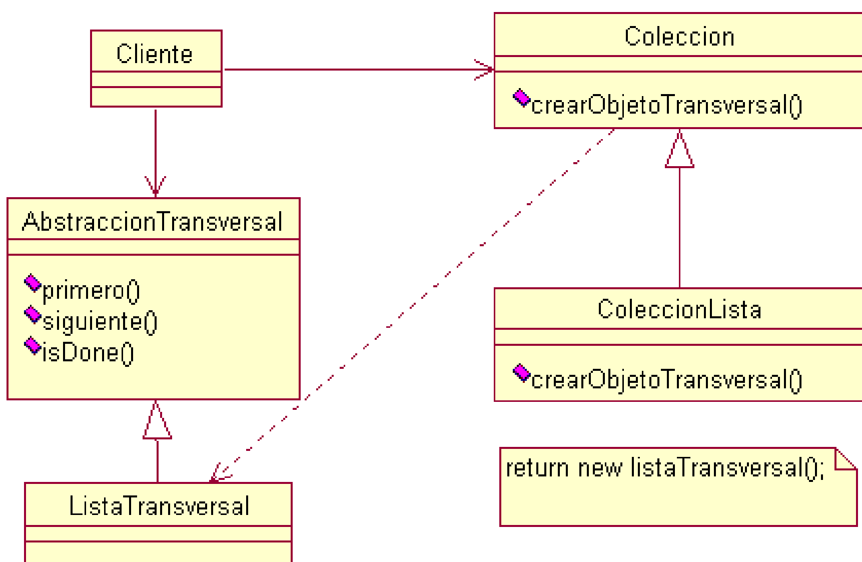


Figura 1.18: Estructura Patrón Iterator

e) Mediator

Nombre: Mediador

Función: Definir un objeto que coordine la comunicación entre otros de distintas clases, pero que funcionan como un conjunto.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.19.

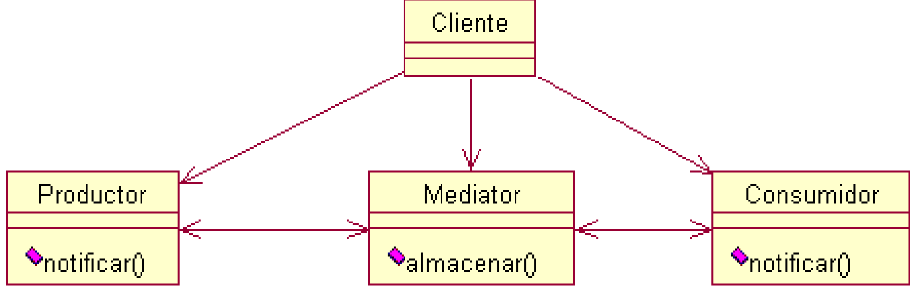


Figura 1.19: Estructura Patrón Mediator

f) Memento

Nombre: Recuerdo

Función: Volver a estados anteriores del sistema.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.20.

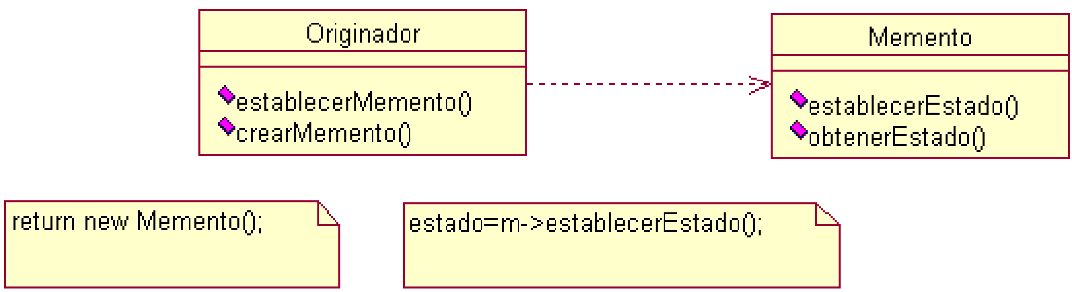


Figura 1.20: Estructura Patrón Memento

g) Observer

Nombre: Observador

Función: Definir una dependencia de uno a muchos entre objetos, de forma que cuando uno cambie de estado se notifique y actualicen automáticamente todos los que dependen de él.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.21.

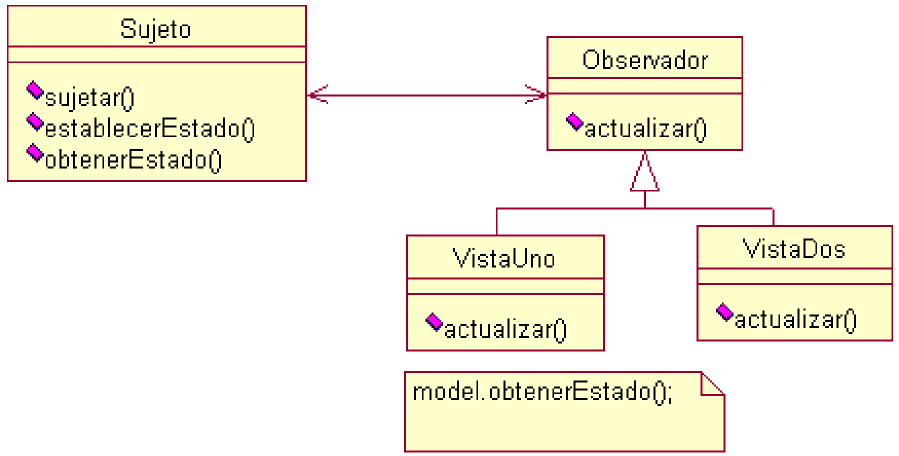


Figura 1.21: Estructura Patrón Observer

h) State

Nombre: Estado

Función: Alterar la conducta interna de un objeto.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.22.

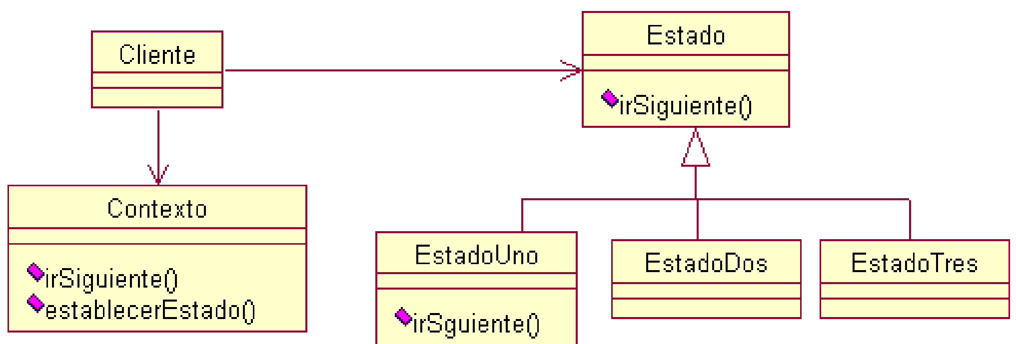


Figura 1.22: Estructura Patrón State

i) Strategy

Nombre: Estrategia

Función: Definir una familia de algoritmos, encapsulados e intercambiables. La estrategia permite el algoritmo variar independientemente de los clientes que lo usan.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.23.

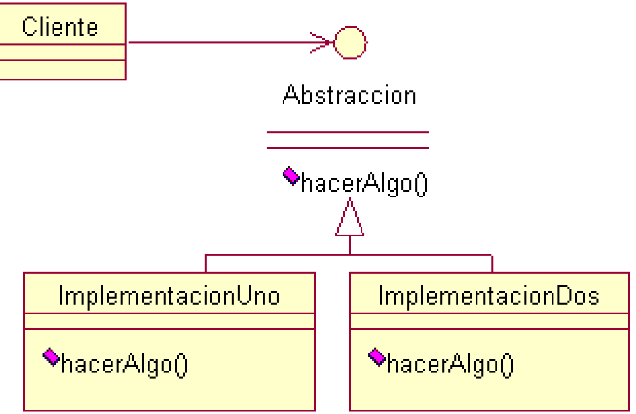


Figura 1.23: Estructura Patrón Strategy

j) Template Method

Nombre: Método de la plantilla

Función: Redefinir subclases siguiendo ciertos pasos de un algoritmo sin cambiar la estructura del mismo.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.24.

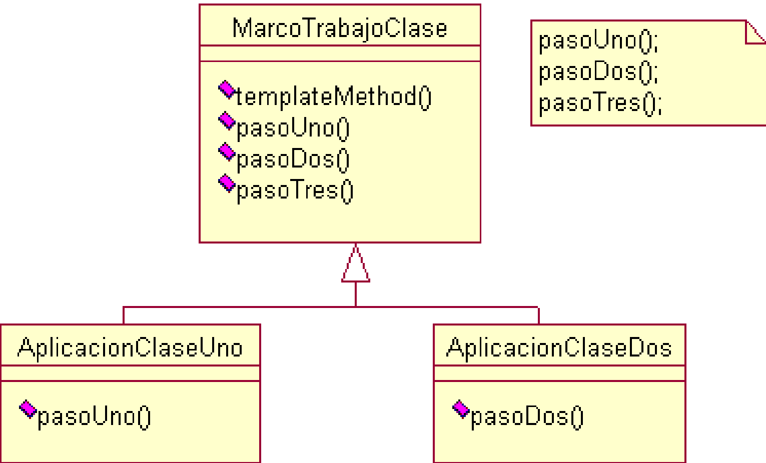


Figura 1.24: Estructura Patrón Template Method

k) Visitor

Nombre: Visitante

Función: Representar un funcionamiento a ser realizado por los elementos de la estructura del objeto, además permite definir un nuevo funcionamiento sin cambiar las clases de los elementos en que opera.

Estructura: La estructura que cumple este patrón se muestra en la Figura 1.25.

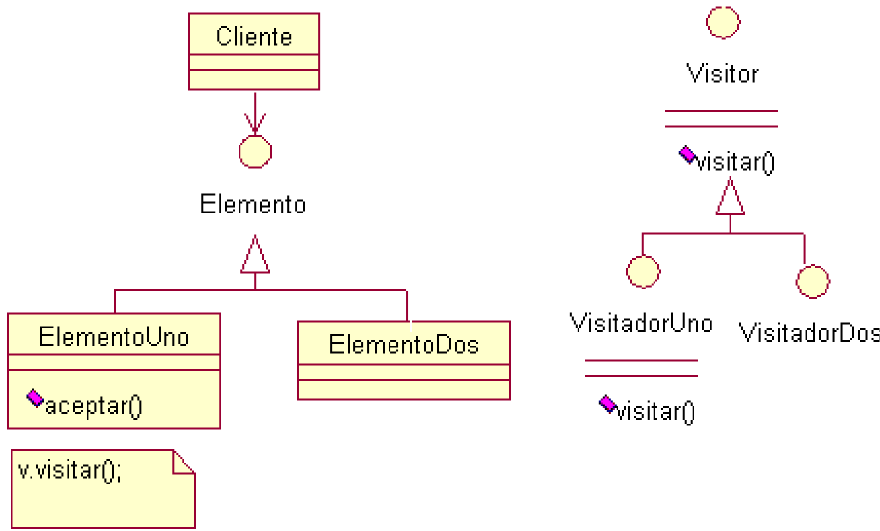


Figura 1.25: Estructura Patrón Visitor

1.2. BENEFICIOS DEL USO DE PATRONES DE DISEÑO

Los patrones de diseño constituyen una poderosa arma para enfrentar los problemas del desarrollo de software, sin embargo, se deben manejar con cautela, pues la inclusión de patrones innecesarios puede llevar al incremento de la complejidad, por lo que para aprovechar al máximo su uso se debe tener muy en cuenta qué patrones se necesita utilizar, y cuáles están asociados con el funcionamiento de éstos, para evitar la introducción de patrones incompatibles entre sí.

La idea del uso de patrones de diseño es la de poder establecer soluciones a problemas recursivos en el desarrollo de software, de manera que se los pueda

utilizar siempre que se presente el mismo problema; para poder cumplir con esto, las soluciones deberán ser bien definidas.

“Nuestro diseño deberá ser específico para el problema que tenemos en frente, pero deberá ser también lo suficientemente generalizado para poder orientarlo a la solución de futuros problemas y requerimientos”[5]

Los patrones de diseño aportan en varios aspectos a mejorar el proceso de desarrollo de software, entre los aportes más importantes debemos destacar:

a) PERMITEN MANTENER UNA ALTA COHESIÓN

Los patrones de diseño permiten que se creen clases de cooperación mutua, pero al mismo tiempo las mantienen lo suficientemente independientes del funcionamiento de otras, por lo cual se obtiene un bajo acoplamiento de los componentes del software y una alta cohesión entre los módulos.

b) FACILITAN LA REUTILIZACIÓN DE SOFTWARE

Este beneficio se deriva de la definición misma de los patrones de diseño, y constituye en una fortaleza para aquellos que los dominan, pues consiste en optimizar el diseño de una solución a un problema de desarrollo determinado, la garantía del uso de los patrones de diseño radica en que se respaldan en la experiencia de la reutilización de los mismo, al enfrentarlos varias veces a problemas similares, pero en entornos diferentes.

---

[5] **Design Patterns, Agosto 1994**

c) FACILITAN LA IDENTIFICACIÓN DE OBJETOS

Permiten identificar abstracciones menos obvias pero más efectivas que aquellas que saltan fácilmente a la vista al momento de diseñar una solución.

d) PERMITEN DISTRIBUIR RESPONSABILIDADES

Al ser organizados de manera jerárquica, en clases y protocolos, los patrones de diseño distribuyen efectivamente las responsabilidades del equipo de desarrollo, pues facilita el desarrollo individual de soluciones que se integran posteriormente al funcionamiento general del sistema.

e) GARANTIZAN REUSABILIDAD, EXTENSIBILIDAD Y MANTENIMIENTO

Esto gracias a que cada patrón permite controlar el tamaño y ubicación de los elementos a ser implementados, garantizando un alto nivel de calidad y simplificando el trabajo en grupo por el vocabulario estándar que se maneja en cada componente, facilitando principalmente el mantenimiento adaptativo (integración de nuevos requerimientos o cambios de los existentes).

f) PROVEEN SOLUCIONES CONCRETAS

Esta podría ser la más importante ventaja del uso de patrones de diseño, ya que estos proveen una solución concreta a un problema haciendo frente a casi todas las metodologías o tecnologías empleadas y además considerando las consecuencias, beneficios y viabilidad de la solución propuesta; esto se debe a que funcionan como una guía para resolver problemas comunes de programación.

g) OFRECEN MAYOR UTILIDAD QUE LOS LENGUAJES DE PROGRAMACIÓN

Para un gran porcentaje de los problemas que se presentan en el desarrollo, los patrones de diseño proveen mayor utilidad que los lenguajes de programación, ya que al ser empleados desde el proceso de diseño, la lógica de la solución está claramente descrita y no da lugar a ambigüedades. Además los patrones son fácilmente aplicables a cualquier lenguaje orientado a objetos.

h) PROVEEN UNA SOLUCIÓN MÁS EQUILIBRADA

Debido a que el empleo de patrones requiere un análisis profundo, se puede encontrar el punto de equilibrio entre las restricciones y objetivos de la solución, obteniendo como resultado un sistema fácil de emplear y que cumple con las expectativas de funcionalidad.

i) FACILITAN EL ENTENDIMIENTO DE LA SOLUCIÓN GRACIAS AL ENCAPSULAMIENTO

Dado que se basan en la experiencia, resulta sencillo encapsular el conocimiento detallado sobre un tipo de problema y sus soluciones; de esta manera es fácil entenderlo sin adentrarse a ámbitos como codificación.

1.3. DESARROLLO DE SOFTWARE CON PATRONES DE DISEÑO VS DESARROLLO SIN PATRONES

1.3.1. DESARROLLO CON PATRONES

El desarrollo de software con patrones de diseño se fundamenta en la mejora de la calidad del sistema generado y las fortalezas; esta forma de desarrollo se basa en las ventajas que ofrecen los patrones de diseño.

Pero cabe recalcar que a pesar de los grandes beneficios que proveen los patrones de diseño, también se deben considerar las debilidades, que si bien es cierto son pocas, estas son lo suficientemente considerables, al punto que pueden anular los beneficios que los patrones ofrecen.

1.3.1.1. Ventajas

a) Capturar las mejores prácticas

Al desarrollar software, el análisis y entendimiento de los diseños exitosos y fallidos es la clave para facilitar la aplicación de los diferentes patrones en cada una de las etapas de desarrollo, ya que se capturan las mejores prácticas y se evita el cometer los mismos errores, reduciendo notablemente la cantidad de código y simplificando la lógica de la solución.

b) Soluciones probadas

El empleo de patrones de diseño en el desarrollo de Software permite obtener soluciones probadas y validadas a un problema particular.

c) Fácil integración

Los sistemas desarrollados con patrones de diseño, además de facilitar la integración de nuevos requerimientos o cambio de los existentes, permiten realizar esto sin afectar la estructura del sistema; la clave para lograrlo es anticiparse a los nuevos requisitos de tal manera que la selección del patrón idóneo asegure una evolución adecuada.

d) Incremento de la eficiencia

Al usar patrones de diseño en el desarrollo de software, el incremento de eficiencia en el diseño es muy notable, ya que ante un problema no es necesario generar una nueva solución: basta con analizar las existentes y tomar las más adecuada con la tranquilidad de saber que estas soluciones son probadas con anterioridad y el riesgo de fallo es prácticamente nulo.

e) Proceso de diseño serio

Emplear patrones en el proceso de desarrollo es el primer paso para iniciar la etapa de diseño de Software serio; en el que se tiene estándares de uso, se facilita el aprendizaje de los componentes ya existentes, todo esto gracias a la documentación de calidad y vocabulario común.

1.3.1.2. Desventajas

a) Incremento de la complejidad debido a la falta de experiencia

Es necesario tener experiencia para reconocer el problema y dar una adecuada solución; es decir que se necesita experiencia en el uso de patrones de diseño para poder utilizarlos efectivamente. En caso de no conocer y determinar correctamente el patrón, se estaría simplemente agregando complejidad al sistema, o peor aún se puede llegar a incluir patrones incompatibles entre sí, lo cual haría imposible integrar los módulos de un sistema, y que se deba desarrollar nuevamente la aplicación desde cero.

b) Mapeo e integración de soluciones

Si bien es cierto que los patrones de diseño describen paso a paso como cubrir un requerimiento o solucionar un problema específico, se debe tener presente que

ningún patrón explica como se lo debe implementar, ni cómo se lo debe integrar al funcionamiento de otros patrones.
Al no contar con una guía o referencia que indique como realizar esto, se puede perder más tiempo y recursos tratando de realizar dicho mapeo e integración, que el que se necesitaría para desarrollar una solución desde cero.

c) Disminución de la capacidad de iniciativa

Al usar componentes, previamente desarrollados, se minimiza la capacidad de iniciativa de las personas y les restan creatividad, ya que se basan en la utilización de patrones conocidos y no se realiza un esfuerzo en buscar nuevas soluciones.

1.3.2. DESARROLLO SIN PATRONES

1.3.2.1. Ventajas

a) Falta de experiencia

La falta de experiencia no es un obstáculo para desarrollar sistemas de calidad, eficaces y eficientes.

b) Reducción del riesgo

Dado que se emplea conceptos conocidos, se optimizan recursos; además se minimizan el tiempo de desarrollo empleando prácticas conocidas y ya probadas con anterioridad en otros proyectos.

1.3.2.2. Desventajas

a) Uso de estándares

Dado que en el proceso de desarrollo de Software común, no se obliga a usar estándares resulta compleja la reutilización de soluciones previas en sistemas similares.

b) Mantenimiento

Dada la falta de documentación adecuada de un sistema se dificulta el mantenimiento adaptativo y correctivo de la solución.

1.3.3. COMPARACIÓN

Como se ha descrito anteriormente, en el desarrollo de Software con patrones y sin patrones presentan tanto ventajas como desventajas; para comprenderlas referirse a la Tabla 1.1.

Se debe tener presente que para asegurar el éxito del desarrollo con patrones, la mayoría de ocasiones será necesario capacitar sobre el uso de patrones de diseño, a los desarrolladores de software; lo cual implica necesariamente incrementos en el presupuesto y plazos del proyecto, es decir mayor riesgo que puede conducir a errores e incluso al fracaso, es por esto que se recomienda iniciar la aplicación de patrones de diseño en proyectos desarrollados por versiones cuya complejidad se incremente de manera controlada, aplicando patrones simples y fáciles de entender por todos los miembros del equipo.

Además es sumamente importante que los desarrolladores de mayor experiencia documenten de manera detallada sus soluciones para que estas puedan ser empleadas por los demás miembros del equipo, facilitando el proceso de análisis y selección de patrones.

Características	**Desarrollo con Patrones**	**Desarrollo sin Patrones**
Uso de mejores prácticas	Alto	Bajo
Facilidad de uso de soluciones probadas	Alto	Bajo
Incremento de eficiencia	Bajo	Bajo
Facilidad de aplicación (experiencia requerida)	Alto	Bajo
Facilidad en el mapeo e integración de módulos	Bajo	Medio
Minimización de Riesgo innecesario	Bajo	Medio
Exige uso de estándares	Alto	Bajo
Facilidad de mantenimiento	Alto	Medio
Fomentar la creatividad	Bajo	Alto

Tabla 1.1: Tabla Comparativa, Desarrollo de software con patrones y desarrollo sin patrones

CAPÍTULO II

2. ESTRUCTURACIÓN DE LA GUÍA

Este capítulo trata sobre la forma de elaborar guías técnicas, se demuestra que no existe esquema único para la creación de guías de ninguna especie, sean estas técnicas, prácticas, metodológicas, etc.; por lo cual es necesario aclarar que una guía se define acorde a su orientación de diseño y aplicación.

2.1. CARACTERÍSTICAS Y FUNCIONES DE LAS GUÍAS DE REFERENCIA

La construcción de guías es una aplicación elemental; pero poderosa de la tecnología informática, ya que el soporte informático es lo que permite obtener un resultado adecuado y muy práctico[6].

Por lo tanto, el desafío es cómo utilizar los recursos que ofrece la tecnología informática, para hacer de las guías verdaderas herramientas de trabajo, útiles y efectivas en su campo de aplicación.

2.1.1. CARACTERÍSTICAS

- Poseen una estructura definida y claramente identificable, misma que se presenta de la siguiente manera:
  - *Introducción:* Información general sobre el tema central de la guía.
  - *Alcance:* Objetivos de su desarrollo y aplicación.
  - *Definiciones*: Conceptos que permitan aclarar términos complejos.
  - *Contenido:* Conjunto de actividades e información necesaria para alcanzar sus objetivos de aplicación.

---

[6] http://www.cmec.ca/international/forum/csep.Chile.ap2.sp.PDF

- Están sujetas a revisión y mejoramiento adaptativo y/o correctivo continuo.
- Pueden ser sencillas, como soluciones a problemas particulares, o altamente detalladas, como soluciones a problemas generales, pero en ningún caso puede ser compleja para su entendimiento y aplicación.
- Permiten la aplicación de su contenido, aún si el usuario cuenta con poca experiencia en el campo de acción de la guía.

2.1.2. FUNCIONES

- Proveer a sus usuarios el conocimiento necesario para enfrentar, de manera eficiente y eficaz, los casos que pueden presentarse en el área de aplicación de la guía.
- Orientar, en términos generales, sobre el campo de acción de la guía.
- Detallar claramente los objetivos de su desarrollo y aplicación, para permitir al usuario conocer los beneficios, de acuerdo a los objetivos que se hayan definido.
- Definir claramente las actividades, y la secuencia de ejecución de las mismas, para alcanzar los objetivos planteados en el desarrollo de la guía.
- Establecer recomendaciones para agilitar el trabajo de sus usuarios.
- Establecer mecanismos de evaluación de la aplicación de la guía, para permitir al usuario medir sus progresos en cuanto al dominio del tema central de la guía.

2.2. DISEÑO DE GUÍAS DE REFERENCIA

Para diseñar guías de referencia, es necesario establecer un esquema de la estructura central de las guías en general, para lo cual resulta útil el análisis y estudio de otras guías; este esquema debe constar de dos componentes fundamentales: fondo y forma[7].

---

[7] **http://www.cmec.ca/international/forum/csep.Chile.ap2.sp.PDF**

a) FONDO:

- El conocimiento o la experiencia con los que se desea fortalecer el dominio del tema central a sus usuarios.
- La transformación didáctica que requieren ese conocimiento y experiencia, para aumentar las probabilidades de éxito de quienes las usen.

b) FORMA:

- La estructura o secuencia que usará para expresar esa estrategia de aplicación a su campo de acción.
- Los aspectos de presentación como lenguaje, tipografía, diagramación, etc.

Estos componentes se deben mantener desde el diseño mismo de la guía, independientemente de la estructura o esquema utilizado para su elaboración. Acorde a las características antes descritas, la guía va a contener una estructura básica, detallada a continuación:

c) INTRODUCCIÓN

Título: Definición del propósito de la guía.
Presentación: Información general sobre la guía, su propósito, orientación de desarrollo, antecedentes, aplicación, y datos adicionales sobre los autores de la misma.

d) ALCANCE

Objetivos: Propósitos de la aplicación de la guía en un caso práctico dentro de su campo de acción. Se redactarán de manera que expresen una acción concreta, de que es lo que pretende hacer la guía al ser aplicada por algún usuario.

Alcance: Define exactamente el campo de acción de la guía mediante el detalle de sus facultades, el alcance debe incluir además las limitaciones de la guía.

e) CONTENIDO

Definiciones: Información adicional sobre el contenido de la guía, puede ser utilizada por el lector para complementar el entendimiento de los términos utilizados en la guía.
Resumen de Contenido: Presentación resumida de los puntos a tratar en la guía, provee al usuario una visión general sobre la guía.
Desarrollo de contenido: Presentación detallada de los puntos y actividades tratadas en la guía para alcanzar los objetivos planteados en el alcance.

2.3. REQUISITOS PARA LA ELABORACIÓN DE GUÍAS DE REFERENCIA[8]

Los requisitos para la elaboración de guías de referencia son:

Conocimiento: Se refiere a aspectos propios de las nociones técnicas manejadas en el diseño y desarrollo de la guía; éstos son:

- Calidad
- Relevancia, es decir si el conocimiento aplicado a la guía es importante y propio de lo que es fundamental en el campo de acción de la guía.
- Pertinencia, es decir si el conocimiento plasmado en la guía es aquello que el lector o usuario de la misma necesita.
- Vigencia, es decir que tan actualizado es el conocimiento a manejarse en la guía.

[8] **Edith Proaño, “Guía para el desarrollo de Aplicaciones Robustas Utilizando Técnicas y Estrategias de Seguridad”**

Estructura: Se refiere a la organización del contenido de la guía, y su correspondencia con el conocimiento y objetivos de diseño y desarrollo de la misma; incluye:

- Factibilidad de las actividades y tareas propuestas en la guía.
- La secuencia de las actividades es la más apropiada para facilitar su aplicación.
- La complejidad del contenido de la guía debe ser baja, lo cual es recomendable para que pueda ser expresada mediante diagramas.

Presentación: Se refiere al papel, la impresión, el tipo de letra, etc. Al tratarse de un proyecto de titulación de la Escuela Politécnica Nacional, la guía está sujeta al estándar de presentación manejado por la misma. Sin embargo es importante considerar la inclusión de contenido didáctico fácil de comprender por el usuario, como diagramas, esquemas, gráficos, etc.

2.4. FACTORES A CONSIDERARSE PARA LA ELABORACIÓN DE LA GUÍA PARA EL USO DE PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE

Para la elaboración de la guía para el uso de patrones de diseño en el desarrollo de software se debe considerar los siguientes puntos:

- Investigar los orígenes, características y aporte, de los patrones de diseño, a las diferentes etapas del desarrollo de software, para poder aprovechar al máximo sus beneficios.
- Determinar las características principales de una guía, las cuales faciliten su entendimiento y aplicación.
- Definir el contenido de la guía de tal manera que los usuarios puedan familiarizarse rápidamente con los patrones de diseño.

- Diseñar la estructura de la guía correctamente, de tal manera que permita aplicar los patrones de diseño de manera óptima, aún cuando el usuario no cuente con mucha experiencia en la aplicación de los mismos.

CAPÍTULO III

3. GUÍA PARA EL USO DE PATRONES DE DISEÑO EN EL DESARROLLO DE SOFTWARE

Para la estructuración de la guía ha sido necesaria la investigación detallada de los patrones, sus orígenes, aplicaciones, ventajas y desventajas, de tal manera que se garantice un uso sencillo sin necesidad de contar con una amplia experiencia en el manejo de patrones de diseño.

3.1. OBJETIVOS

- Proveer una importante herramienta que permita comprender y tomar decisiones adecuadas, para el uso de patrones de diseño en el desarrollo de software.
- Describir el funcionamiento de los principales patrones de diseño y ubicarlos, de acuerdo a su aporte, en la solución de problemas de las diferentes etapas del proceso de desarrollo de Software.
- Brindar una serie de pasos que garanticen el éxito al aplicar los patrones de diseño y faciliten el desarrollo de aplicaciones.

3.2. ALCANCE

La presente guía describe el funcionamiento y uso de los patrones de diseño en la implementación de aplicaciones de software, de manera clara y precisa; además, provee una serie de pasos a seguir para la implementación exitosa de los patrones en sistemas de Software, tratando de obtener soluciones reusables.

Sin embargo la guía no ofrece un estándar de implementación de patrones, sino una base que describe la identificación, organización y aplicación de patrones, con el fin de entender cómo pueden ser empleados los patrones en el desarrollo de software

de una manera sencilla, obteniendo los mejores resultados en la solución de problemas recursivos.

Además la guía no excluye la posibilidad de enfoques alternativos que ofrezcan más ventajas o facilidades en el proceso de aprendizaje y uso de los patrones de diseño para solucionar problemas puntuales del desarrollo de software.

3.3. DEFINICIONES

- ***Campo de acción "Objetos":*** Cuando el patrón de diseño es utilizado en la implementación de una clase.
- ***Campo de acción "Clases":*** Cuando el patrón de diseño es utilizado en la definición de una clase.
- ***Acoplamiento[9]:*** Grado de interdependencia entre los componentes de un sistema informático (módulos, funciones, subrutinas, bibliotecas, etc.); es decir, define el nivel de dependencia entre los elementos del sistema, mientras menor sea el acoplamiento, mayor será la calidad del software.
- ***Cohesión[10]:*** Define la forma en que se agrupan unidades de software (módulos, subrutinas, funciones, etc.) en una unidad mayor. Por ejemplo: la forma en que se agrupan funciones en una biblioteca de funciones o la forma en que se agrupan métodos en una clase, mientras mayor sea la cohesión, mejor será la calidad del software.
- ***Escalabilidad[11]:*** Propiedad deseable en un sistema, que indica su habilidad para poder hacerse más grande (en tamaño o funcionalidad) sin perder calidad en sus servicios, requiere un pensamiento cuidadoso desde el principio del desarrollo.
- ***Mantenimiento[12]:*** Consiste de un conjunto de tareas necesarias para asegurar el buen funcionamiento de un aplicativo, el mantenimiento se clasifica en preventivo, correctivo y complementario.

---

[9] http://www.wordreference.com/definicion/acoplamiento
[10] http://www.wordreference.com/definicion/cohesión
[11] http://www.wordreference.com/definicion/escalabilidad

- ***Reutilización (Reutilización de código)[13]:*** Consiste en el uso de software existente para desarrollar un nuevo software; la idea es que parte, o todo el código, de un programa de computadora escrito una vez sea, o pueda ser usado, en otros programas. La reutilización de códigos programados es una técnica común que intenta ahorrar tiempo y energía, reduciendo el trabajo redundante.
- ***Dinamismo[14]:*** Propiedad deseable en un sistema, que indica su habilidad para poder modificar su comportamiento o funcionalidad dinámicamente, sin necesidad de ser sometido a mantenimiento.
- ***Redundancia[15]:*** Consiste de la repetición innecesaria de código, generalmente repetición de funciones, rutinas o algoritmos, etc. que podrían ser codificadas una sola vez y utilizadas desde otros elementos.
- ***Granularidad[16]:*** Es la especificidad a la que se define un nivel de detalle en una tabla, es decir, si hablamos de una jerarquía la granularidad empieza por la parte más alta de la jerarquía hasta el nivel más bajo.

3.4. CONTENIDO

I INTRODUCCIÓN

Esta guía ha sido desarrollada para poder facilitar el entendimiento de los patrones de diseño, y para ayudar a utilizarlos de manera fácil y óptima en el desarrollo de software.

La guía describe una serie de pasos que permiten al equipo de desarrollo orientar todo el proceso de construcción del software al uso de patrones; además se incluye un catálogo de los primeros patrones de diseño descritos en la ingeniería de software.

---

[12] http://www.wordreference.com/definicion/mantenimiento
[13] http://www.wordreference.com/definicion/reutilización
[14] http://www.wordreference.com/definicion/dinamismo
[15] http://www.wordreference.com/definicion/redundancia
[16] http://es.wikipedia.org/wiki/Granularidad

Los pasos de la guía son: caracterización del sistema, modelado de la solución, análisis detallado, arquitectura de la solución, diseño de la solución y codificación del sistema.

II GUÍA PARA EL USO DE PATRONES

En esta sección se describen los pasos para poder utilizar patrones de diseño en el desarrollo de software, las secciones incluidas contienen un catálogo de patrones y recomendaciones para realizar el desarrollo de software utilizándolos correctamente.

1. Catálogo de Patrones

Esta sección presenta los patrones de diseño descritos en el libro “Design Patterns, elements of reusable object-oriented software”, cubriendo los siguientes aspectos básicos en cada uno de estos:

Descripción: son los aspectos básicos que definen a cada patrón.

- ***Objetivo:*** describe el propósito general que define al patrón de diseño.
- ***Problema:*** es la manera como normalmente es implementado un método que satisface una necesidad o requerimiento específico.
- ***Solución:*** es la manera sencilla y eficiente de implementar un método que satisface una necesidad o requerimiento específico.
- ***Campo de acción:*** es el nivel en el que se recomienda emplear el patrón; el cual puede ser clase u objeto.
- ***Diagrama o Implementación:*** es la representación de un patrón ya sea mediante el diagrama de clases o bien el código de implementación del mismo.

Aplicación: son casos específicos en los que los patrones pueden de mucha ayuda; por lo cual se recomienda su implementación.

Colaboraciones: son patrones relacionados que pueden implementarse de manera conjunta para dar solución a un problema específico.

1.1. Patrones de Creación

Los patrones de creación proporcionan ayuda a la hora de crear objetos, principalmente cuando esta creación requiere tomar decisiones. Esta toma de decisiones puede ser dinámica.

1.1.1. Abstract Factory

DESCRIPCIÓN:

- **Objetivo:** Permitir la creación de grupos de objetos relacionados o dependientes, sin especificar sus clases concretas.
- **Problema:** Para implementar grupos de objetos es necesario especificar sus nombres exactos en la clase cliente, es decir que en caso de que se necesite modificar un objeto también se debe cambiar la clase que lo instancia.
- **Solución:** El patrón de diseño Abstract Factory aísla a las clases “clientes” de los nombres y definiciones de las clases “producto” (soporta la creación implícita de objetos); de manera que, la única forma de conseguir un “producto” es a través de una clase Abstract Factory, de esta manera el conjunto de productos pueden modificarse fácilmente sin necesidad de actualizar cada clase cliente.

  Una clase Abstract Factory puede especificarse en varias clases concretas, de manera que cada una de ellas pueda crear diferentes tipos de productos y en diferentes combinaciones.
- **Campo de acción:** aplicado a nivel de objeto.

- **Diagrama:**

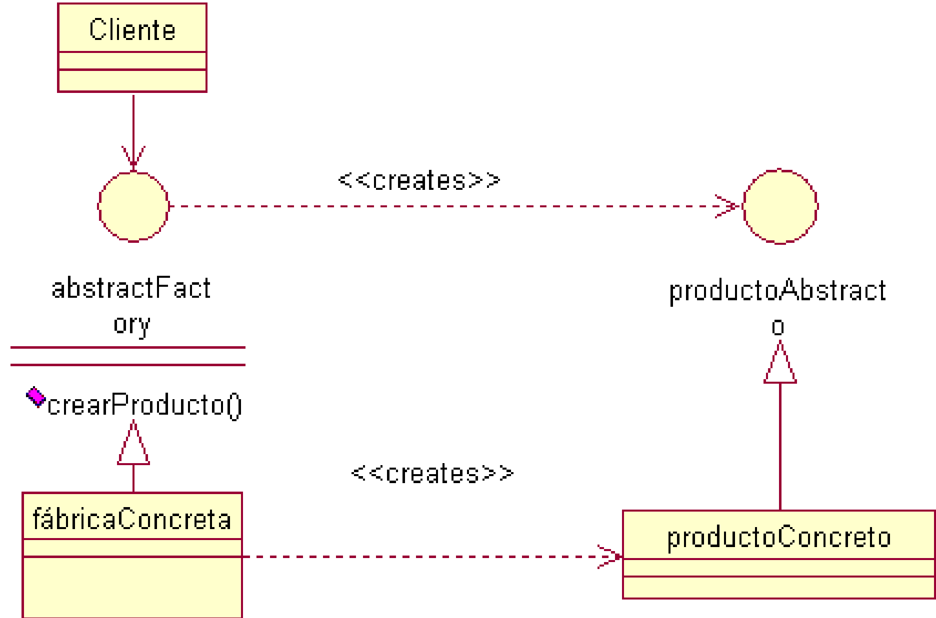


Figura 3.1: Patrón Abstract Factory

A continuación se explica cada elemento de la Figura 3.1:

Fábrica Abstracta: Declara una interface para las operaciones que crea objetos de productos abstractos.

Fábrica Concreta: Implementa las operaciones para crear objetos de productos concretos.

Producto Abstracto: Define un objeto producto para ser creado por la correspondiente fábrica concreta.

APLICACIÓN:

El uso del patrón Abstract Factory se recomienda en los siguientes casos:

- El sistema a desarrollarse debe ser independiente de la forma en que se crean sus objetos.
- El sistema debe ser configurado con uno o varios conjuntos de productos posibles.
- Un grupo de objetos relacionados es diseñado para usarse en conjunto, y es necesario reforzar esta restricción.

- Se desea implementar una librería de clases de productos, y solo se desea revelar sus interfaces, no sus implementaciones.

COLABORACIONES:

- Una fábrica abstracta siempre es un objeto de instancia única, correspondiente al patrón Singleton.
- Se puede implementar una fábrica abstracta como un patrón Factory Method o un patrón Prototype.

1.1.2. Builder

DESCRIPCIÓN:

- **Objetivo:** Separar la construcción de un objeto complejo de su representación, de manera que se puedan crear diferentes representaciones de dicho objeto utilizando el mismo proceso de construcción.
- **Problema:** Para emplear un mismo proceso de construcción para diferentes objetos, se debe realizar modificaciones el algoritmo o estrategia de construcción; lo cual incrementa la complejidad y reduce la modularidad del sistema.
- **Solución:** El patrón de diseño Builder permite ubicar la lógica de construcción de un objeto fuera del mismo y programarla en una clase independiente (Constructor), la cual devolverá al cliente una representación determinada del objeto sin necesidad de que éste o la clase cliente sepan la secuencia de creación de dicho objeto.
- **Campo de acción:** aplicado a nivel de objeto.

- **Diagrama:**

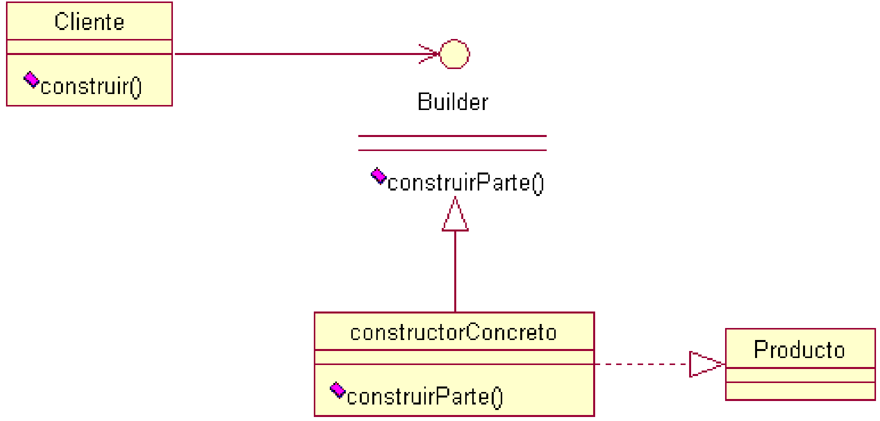


Figura 3.2: Patrón Builder

A continuación se explica cada elemento de la Figura 3.2:

Builder: Declara una interface para la creación de partes de un objeto.

Cliente: Construye un objeto mediante la interfaz Builder.

Constructor Concreto: Construye y ensambla las partes del objeto; es decir se encarga de construir la representación interna del producto y definir el proceso con el cual se ensambla.

Producto: Representa el objeto complejo en construcción

APLICACIÓN:

El uso del patrón Builder es recomendable cuando:

- El algoritmo de creación de objetos complejos debe ser independiente de las partes que conforman al objeto y de la forma en que ellas son ensambladas.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.
- Se necesita tener un control minucioso sobre el proceso de creación de un objeto.

COLABORACIONES:

- El objeto construido mediante un patrón Builder generalmente es un objeto COMPOSITE.

1.1.3. Factory Method

DESCRIPCIÓN:

- **Objetivo:** Definir una interfaz para la creación de un objeto, permitiendo a las subclases decidir cual clase instanciar; es decir que el patrón de diseño Factory Method permite a una clase asumir la instanciación de una subclase.
- **Problema:** Para permitir que una clase instancie una subclase se debe especificar claramente los nombres de los objetos a ser creados dependiendo del estado de la aplicación; esto incrementa la complejidad y dificulta el mantenimiento de la aplicación
- **Solución:** El patrón de diseño Factory Method permite separar la lógica de creación de objetos de la clase cliente; de tal manera que los parámetros enviados por dicha clase son los que definen el objeto; es decir que se puede crear objetos con funcionalidad diferente a través de la misma interfaz.
- **Campo de acción:** aplicado a nivel de clases.
- **Diagrama:**

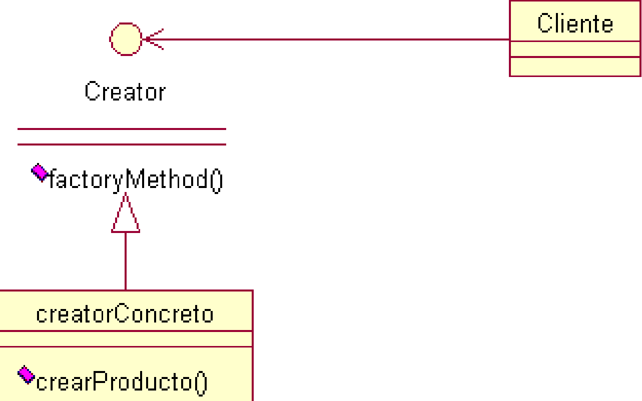


Figura 3.3: Patrón Factory Method

A continuación se explica cada elemento de la Figura 3.3:

Creador: Declara el método que retorna un objeto producto.

Creador Concreto: Retorna una instancia para un producto.

Cliente: utiliza la interfaz Creador

APLICACIÓN:

Se recomienda el uso del patrón Factory Method cuando:

- Es importante la flexibilidad del sistema.
- Una clase es incapaz de anticipar que objetos debe implementar.
- Se requiere manejar varios objetos de manera similar, pero que los resultados del uso de los mismos sea diferente.

COLABORACIONES:

- Un patrón Factory Method generalmente implementan patrones Abstract Factory.
- Generalmente son llamados dentro de un patrón Template Method.

1.1.4. Prototype

DESCRIPCIÓN:

- **Objetivo:** Especificar los tipos de objetos a ser creados usando una instancia Prototype, y crear los nuevos objetos copiando dicho prototipo.
- **Problema:** Para crear varios objetos, a pesar de que estos poseen una misma estructura básica, se debe configurar cada clase en la que serán empleados; es decir que cada objeto debe ser programado de manera independiente consumiendo muchos recursos y complicando la codificación.
- **Solución:** El patrón Prototype crea un objeto prototipo, conocido como administrador, que se encarga de crear o destruir dinámicamente varios objetos prototipos, manteniendo registros de los disponibles; es decir que la clase cliente solo se encarga de consultar al administrador si existe un registro a partir del cual pueda clonar el número de objetos necesarios, haciendo uso de la operación “Clone”, considerando que los prototipos clonados que tienen estructura compleja requieren una copia más detallada porque debe ser independiente del original.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

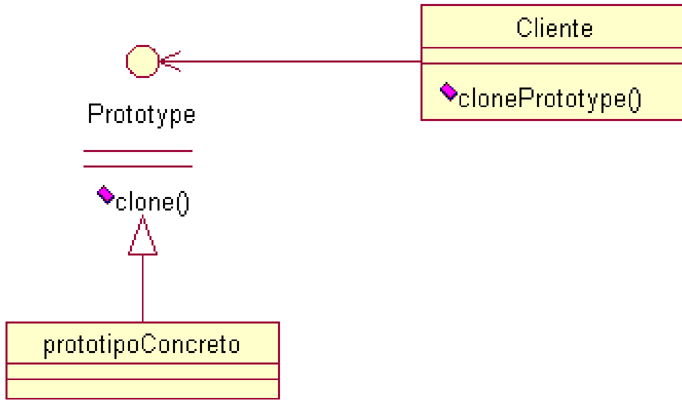


Figura 3.4: Patrón Prototype

A continuación se explica cada elemento de la Figura 3.4:

Prototype: Declara una interface para ser clonada.

***Prototipo Concreto**:* **I**mplementa una operación para clonarse.

Cliente: Crea un nuevo objeto que se encarga de consultar al prototipo para clonarlo.

APLICACIÓN:

Se recomienda el uso del patrón Prototype cuando:

- El sistema debe ser independiente de la manera en la que sus productos son creados, compuestos y representados.
- Las clases a ser instanciadas se especifican en tiempo de ejecución, es decir dinámicamente.
- Las instancias de una clase pueden tener solo una de las pocas combinaciones de estado; lo cual puede ser más conveniente para instalar un número correspondiente de prototipos.

COLABORACIONES:

- El patrón Prototype puede ser usado en conjunto en el patrón Abstract Factory; en la mayoría de los casos este almacena un conjunto de prototipos para realizar la clonación.

- Los diseños que hacen pesado el uso de los patrones Composite y Decorator, suelen ser muy útiles y benefician al patrón Prototype.

1.1.5. Singleton

DESCRIPCIÓN:

- **Objetivo:** Garantizar la creación de una instancia única para una clase dada.
- **Problema:** Para garantizar la creación de una sola instancia se requiere altos niveles de validación y una variable global; la cual es difícil de controlar cuando se presentan múltiples instanciaciones de objetos.
- **Solución:** El patrón Singleton crea una clase que instancia el único objeto que será responsable de la creación, inicialización y acceso; dicha instancia debe ser un dato privado, ya que se debe ocultar la operación de creación de la instancia. Además se requiere una función estática pública que se encargue del encapsulamiento de la inicialización y proporcione un punto de acceso global. En el proceso de implementación del patrón se debe garantizar que cada clase es responsable de controlar que no se permita más de una instancia.
- **Campo de acción:** aplicado a nivel de objetos y clases.
- **Diagrama:**

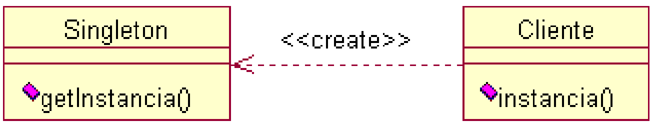


Figura 3.5: Patrón Singleton

A continuación se explica cada elemento de la Figura 3.5:

Singleton: Implementa la instancia única.

Cliente: Define una operación "Instancia" para que el cliente pueda acceder por instancia única.

APLICACIÓN:

Se recomienda el uso del patrón Singleton cuando:

- El sistema exige exactamente una instancia de una clase, la cual debe ser accesible para los clientes desde un punto de acceso bien definido.
- La única instancia debe ser extendida a subclases y los clientes deben ser capaces de usarla sin modificar su código.

COLABORACIONES:

- Un patrón Singleton es frecuentemente relacionado y empleado en la implementación del patrón Abstract Factory, sobre todo cuando se trata de una fábrica concreta.

1.2. Patrones Estructurales

Los patrones estructurales proporcionan ayuda en la combinación de clases y los objetos dando lugar a estructuras más complejas.

1.2.1. Adapter

DESCRIPCIÓN:

- **Objetivo:** Permite adaptar o modificar, una interfaz existente de tal manera que a clases, que de otra forma serían incompatibles, puedan interactuar.
- **Problema:** Para hacer uso de un objeto, cuya interfaz resulta incompatible; es necesario crear una nueva o en caso contrario modificar la aplicación para que pueda integrarse dicha interfaz; en cualquiera de los dos caminos el incremento de la complejidad es notable.
- **Solución:** El patrón de diseño Adapter permite adaptar la interfaz original de un objeto con una compatible con el objeto esperado por la clase cliente; es decir que el patrón es visto como una capa delgada de código entre dos objetos llamados “sintácticamente incompatibles”.

- **Campo de acción:** aplicado a nivel de clases.
- **Diagrama:**

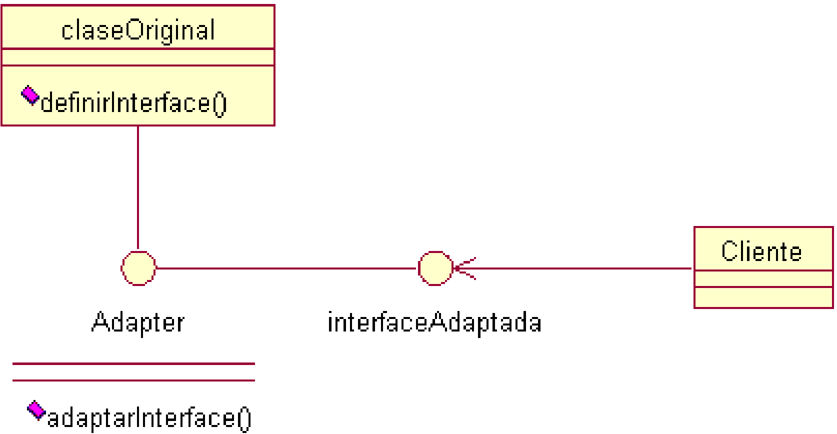


Figura 3.6: Patrón Adapter

A continuación se explica cada elemento de la Figura 3.6:

Clase original: define la interface a ser adaptada.

Adapter: adapta a la interface original para que pueda ser accedida por el cliente.

Interface Adaptada: representa la capa de código que permite interactuar a los objetos incompatibles.

Cliente: instancia el objeto que hará uso de la interface adaptada.

APLICACIÓN:

El uso del patrón Adapter se recomienda en los siguientes casos:

- Se necesita utilizar una clase existente a través de una interfaz incompatible con dicha clase.
- Se requiere crear una clase reutilizable, que coopere con clases no relacionadas o no previstas; es decir, clases que no necesariamente tienen interfaces compatibles.
- Se necesita soportar diferentes conjuntos de métodos para diferentes propósitos.

- Cuando las incompatibilidades a mitigar son diferencias de sintaxis, el método que se está adaptando aún necesita ser capaz de ejecutar la tarea adaptada, si esto no es posible, el patrón Adapter no solucionará la incompatibilidad.
- Se desee permitir al usuario seleccionar entre diferentes GUI's (Interfaces Gráficas de Usuario), para explotar el mismo sistema de la forma que le resulte más cómoda al usuario.

COLABORACIONES:

- Ninguna.

1.2.2. Bridge

DESCRIPCIÓN:

- **Objetivo:** Separar una abstracción de su implementación, de manera que las dos puedan variar independientemente.
- **Problema:** Para hacer uso de diferentes representaciones de una clase abstracta, se requiere implementar una herencia ya que una clase abstracta define la interfaz para dicha abstracción y las clases concretas, o herederas, implementan la interfaz de diferentes formas, sin embargo esta implementación es limitante, considerando que se enlaza permanentemente a la abstracción, lo cual dificulta el modificar, extender, y reutilizar las abstracciones y las implementaciones independientemente.
- **Solución:** El patrón de diseño bridge permite conectar mediante un puente una abstracción de sus diferentes implementaciones, lo cual crea una jerarquía para las abstracciones y otra para las implementaciones, permitiendo gestionar ambas jerarquías de clases independientemente.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

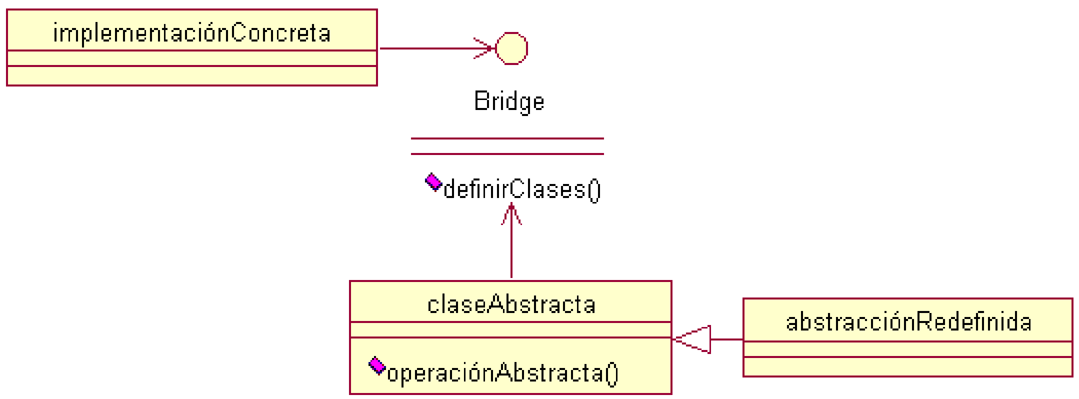


Figura 3.7: Patrón Bridge

A continuación se explica cada elemento de la Figura 3.7:

Clase *Abstracta:* define la operación que permite crear abstracciones.

Bridge: define una interface para la implementación de las clases.

Abstracción refinada: extiende la clase definida para una abstracción específica.

Implementación Concreta: define una implementación concreta.

APLICACIÓN:

El uso del patrón Bridge se recomienda en los siguientes casos:

- Se necesite suprimir un enlace permanente entre una abstracción y su implementación, por ejemplo, cuando se debe enlazar o cambiar una implementación en tiempo de ejecución.
- La abstracción y la implementación deben ser extensibles mediante la agregación de subclases. En este caso el patrón de diseño bridge permite combinar las diferentes abstracciones e implementaciones y extenderlas independientemente.
- Los cambios en la implementación de una abstracción, no deben impactar a las clases clientes, es decir, su código no debe ser re-compilado.
- La necesidad de implementar varias representaciones de una abstracción puede generar una proliferación de clases.
- Se desea compartir una implementación entre múltiples objetos, y que este hecho sea transparente para el cliente.

COLABORACIONES:

- El patrón Abstract Factory puede crear y configurar un patrón Bridge determinado.

1.2.3. Composite

DESCRIPCIÓN:

- **Objetivo:** Conformar estructuras jerárquicas de manera que componentes individuales puedan ser tratados al igual que grupos de componentes. Las operaciones típicas para componentes incluyen agregar, eliminar, desplegar, encontrar y agrupar.
- **Problema:** Para crear objetos, individuales o una agrupaciones, semejante a una estructura de árbol con nodos compuestos o simples; se debe hacer uso de listas enlazadas, sin embargo, la dificultad en estos casos es saber reconocer un objeto compuesto o uno simple, para poder determinar que operaciones aplicar a cada caso.
- **Solución:** El patrón de diseño composite permite referenciar objetos individuales u hoja, empleando una interface “Componente” misma que ejecuta directamente la operación en caso de ser un objeto hoja, mientras que en caso de ser un objeto compuesto replica todas las operaciones a cada uno de los objetos componentes u hoja.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

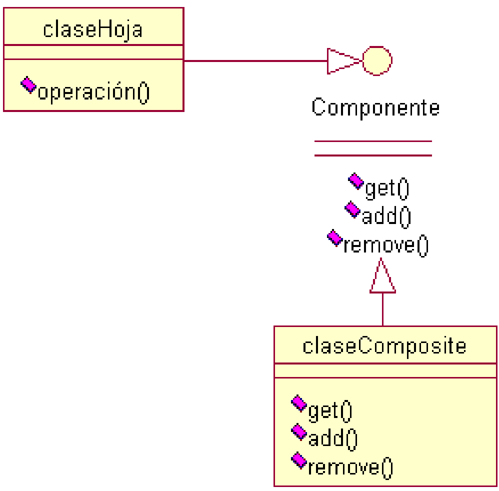


Figura 3.8: Patrón Composite

A continuación se explica cada elemento de la Figura 3.8:

Interface Componente: declara la interface para los objetos de la composición.

Clase Composite: almacena los componentes hijos.

ClaseHoja (Componente): representa los objetos hoja en la composición.

APLICACIÓN:

El uso del patrón Composite se recomienda en los siguientes casos:

- Se desea representar jerarquías completas de objetos o simplemente una parte de ellas
- Se desea que las clases clientes manejen uniformemente a todos los objetos de una estructura jerárquica, desconociendo si están gestionando un objeto simple o uno compuesto.

COLABORACIONES:

- Un composite generalmente es ideal para aplicar una cadena de responsabilidades Chain of Responsability.
- Un patrón decorator generalmente usa un composite.

- El patrón iterador puede ser utilizado para recorrer los composites.

1.2.4. Decorator o Wrapper

DESCRIPCIÓN:

- **Objetivo:** Implementar responsabilidades adicionales a un objeto, de manera dinámica, y proporcionar una alternativa flexible para extender la funcionalidad de una subclase.
- **Problema:** Para agregar una conducta o estado específico a los objetos de manera individual y en tiempo de ejecución, se requiere implementar la herencia; sin embargo esta se aplica a toda una clase de manera estática y el cliente pierde el control.
- **Solución:** El patrón Decorator conforma la interface del componente que trabaja como frontera entre una clase y sus respectivas subclases, misma que es transparente a los clientes, la clase decorator remite el pedido al componente y realiza acciones adicionales; mientras que la transparencia permite anidar recursivamente decoradores de tal manera que se satisfaga un número ilimitado de responsabilidades agregadas dinámicamente.
- **Campo de acción:** aplicado a nivel de objetos.
- **Diagrama:**

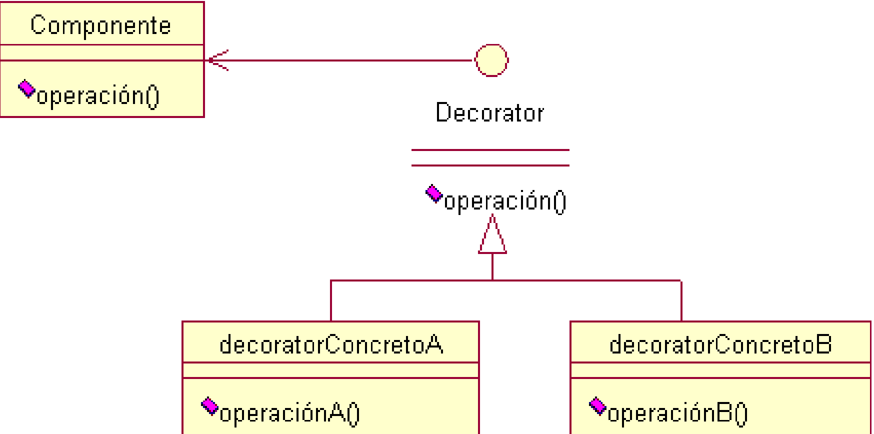


Figura 3.9: Patrón Decorator

A continuación se explica cada elemento de la Figura 3.9:

Componente: define la interface para los objetos que pueden tener responsabilidades añadidas dinámicamente.

Decorator: mantiene una referencia a un objeto Component y define una interface que conforma la interface Component.

Decorator Concreto: añade responsabilidades a los componentes.

APLICACIÓN:

Se recomienda el uso del patrón Decorator cuando:

- El sistema exige agregar responsabilidades dinámica y transparentemente a los objetos individuales; es decir sin afectar a los otros objetos.
- Se requiere retirar responsabilidades sin afectar el funcionamiento.
- La extensibilidad a subclase es impráctica. A veces un gran número de extensiones independientes es posible y produciría una explosión de subclases para dar soporte a cada combinación.

COLABORACIONES:

- Un patrón Decorator es normalmente considerado un patrón Composite generado con solo un componente; ya que un Decorator añade responsabilidades adicionales.
- Los patrones Decorator y Strategy trabajan en conjunto, en el ámbito de variación de un objeto; el uno se encarga de su presentación mientras que el otro de su forma, respectivamente.

1.2.5. Facade

DESCRIPCIÓN:

- **Objetivo:** Proveer una interface unificada para un conjunto de interfaces de un subsistema, facilitando el uso de dicho subsistema.

- **Problema:** La implementación de una solución que simplifique el uso de un subsistema complejo se la lleva a cabo mediante un proceso de encapsulamiento, mismo que en muchas ocasiones resulta no ser el mejor camino.
- **Solución:** El patrón Facade introduce un objeto que proporciona una única interface simplificada minimizando la comunicación y dependencias entre los subsistemas. El objeto realiza una tarea compleja ya que se debe encargar de traducir la interface de cada subsistema a un lenguaje común para permitir la interacción entre ellas.
- **Campo de acción:** aplicado a nivel de objetos.
- **Diagrama:**

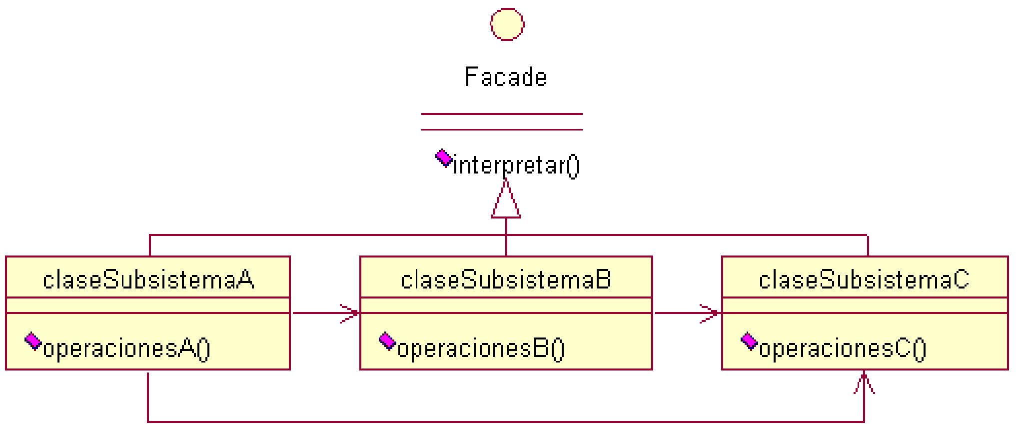


Figura 3.10: Patrón Facade

A continuación se explica cada elemento de la Figura 3.10:

Facade: conoce cual subsistema es responsable de determinada repuesta para el cliente.

ClasesSubsistemas: implementa la funcionalidad del subsistema.

APLICACIÓN:

Se recomienda el uso del patrón Facade cuando:

- Evitar la creciente complejidad de los subsistemas debido a su proceso de evolución, mediante la implementación de una interface simple

- Se presentan demasiadas dependencias entre los clientes y la implementación de clases, facilitando la portabilidad.
- El objetivo es definir un punto de entrada para cada subsistema, para simplificar la interacción entre subsistemas mediante sus respectivas implementaciones del patrón Facade.

COLABORACIONES:

- Los patrones Facade y Abstract Factory pueden trabajar en conjunto para proveer una interface para crear los objetos del subsistema de una manera independiente.
- Los patrones Facade y Mediator son similares en que abstraen la funcionalidad de clases existentes, considerando que el propósito de Mediator es resumir la comunicación entre los objetos colegas.

1.2.6. Flyweight

DESCRIPCIÓN:

- **Objetivo:** Eliminar o reducir la redundancia cuando hay gran cantidad de objetos que contienen información idéntica, además lograr un equilibrio entre flexibilidad y rendimiento.
- **Problema:** La aplicación requiere instanciar objetos iguales, con funcionalidad diferente, lo cual es altamente costoso en términos de memoria.
- **Solución:** El patrón Flyweight permite compartir objetos de tal manera que se emplee una granularidad fina sin los altos costos de implementación de la misma; para lo cual cada objeto es dividido en dos partes: estado-independiente o intrinsic, almacenada en el objeto Flyweight; y estado-dependiente o extrinsic, almacenada en los objetos clientes que se encargan de pasar a Flyweight sus operaciones cuando estas son invocadas.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

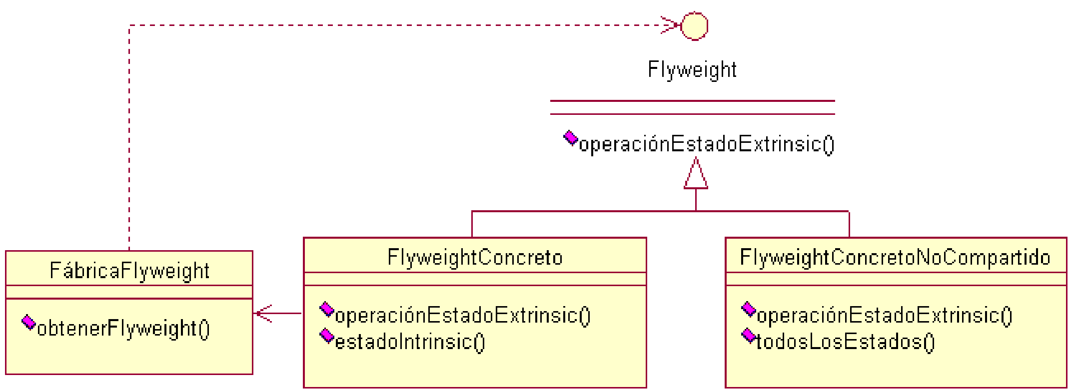


Figura 3.11: Patrón Flyweight

A continuación se explica cada elemento de la Figura 3.11:

Flyweight: declara una interfaz a través de la cual los flyweights pueden recibir y actuar sobre los estados no compartidos.

Fábrica Flyweight: garantiza que los objetos flyweight se comparten de forma apropiada. Cuando un cliente solicita un flyweight, el objeto de la clase FlyweightFactory proporciona una instancia existente, o crea una.

Flyweight Concreto: Implementa la interfaz Flyweight y almacena los estados compartidos, si los hay. Un objeto ConcreteFlyweight debe ser compartible. Cualquier estado que almacene debe ser intrínseco; es decir, debe ser independiente de su contexto.

Flyweight Concreto No Compartido: permite que se comparta los objetos. Es común que los objetos de esta clase tengan hijos de la clase ConcreteFlyweight en algún nivel de su estructura.

APLICACIÓN:

Se recomienda el uso del patrón Flyweight cuando:

- El sistema requiere el uso de grandes números de objetos; lo cual produce altos costos de almacenamiento.

- La mayoría de grupos de objetos pueden ser reemplazados por pocos objetos compartidos.
- El sistema no depende de la identidad del objeto.

COLABORACIONES:

- El patrón Flyweight trabaja en conjunto con Composite, para representar una estructura jerárquica mediante un gráfico de nodos.
- Tanto los patrones State y Strategy trabajan de manera idónea con objeto Flyweight.

1.2.7. Proxy O Sustituto

DESCRIPCIÓN:

- **Objetivo:** Proveer un sustituto para un objeto de tal manera que se pueda controlar el acceso a este.
- **Problema:** Para dar soporte a los objetos en el instante en que son requeridos, se debe instanciar previamente a todos los objetos de tal manera que estos se encuentren listos para que el cliente haga uso de los mismos; sin embargo implica un consumo alto de recursos.
- **Solución:** El patrón Proxy permite controlar el acceso a un objeto instanciándolo en el momento en el que el objeto sea usado realmente; para lo cual emplea un objeto imagen “proxy” que siempre está presente como representación del original, y se encarga de instanciarlo solo cuando es requerido; de tal manera que se pone a disposición una referencia más versátil y sofisticada que un simple punto de acceso a un objeto; mediante tres opciones: un “proxy remoto” ocultando el hecho de que un objeto reside en un espacio diferente o “proxy virtual” optimizando la creación de un objeto en el momento que es requerido o de ambas maneras empleando referencias inteligentes realizando tareas adicionales cuando el objeto es accedido.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

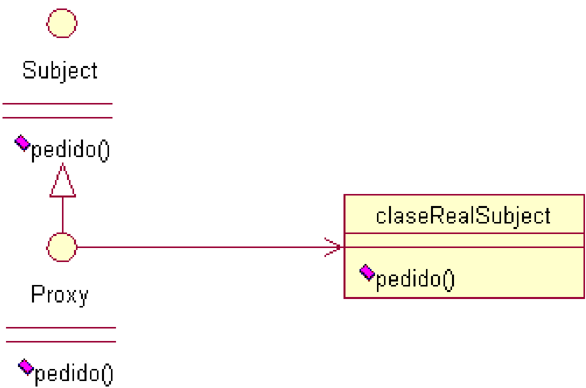


Figura 3.12: Patrón Proxy

A continuación se explica cada elemento de la Figura 3.12:

Proxy: provee una interface que permite acceder a la clase que define el objeto real.

Subject: define la interface común para la interface Proxy y la clase que define el objeto real.

ClaseRealSubject: define el objeto real que es representado por el objeto Proxy.

APLICACIÓN:

Se recomienda el uso del patrón Proxy cuando:

- El sistema requiere una representación remota para un objeto en un diferente lugar.
- Se desea ocultar la complejidad de un objeto representándolo con una simple que no requiera mayor conocimiento para de esta manera facilitar su uso.
- Los objetos deben tener diferentes puntos de acceso, controlando el acceso al objeto original, sin que esto signifique instanciar el objeto en diferentes lugares.

COLABORACIONES:

- Tanto el patrón Proxy como el Decorator tienen aplicaciones similares; sin embargo se debe tener cuidado ya que tienen propósitos diferentes, es decir que un Decorator agrega una o más responsabilidades a un objeto mientras que un Proxy controla el acceso al objeto.

1.3. Patrones de Comportamiento

Estos patrones de diseño están relacionados con algoritmos y asignación de responsabilidades a los objetos.

1.3.1. Chain Of Responsability

DESCRIPCIÓN:

- **Objetivo:** Evitar acoplar a la clase solicitante de una acción con la clase ejecutor de la misma para permitir a más de un objeto manejar la solicitud. Encadenar los objetos receptores y pasar la demanda a lo largo de toda la cadena hasta que un objeto la maneje.
- **Problema:** Una aplicación necesita ejecutar una acción independientemente del estado de la misma, convencionalmente es posible cubrir este requerimiento, pero para lograrlo, cada objeto debe instanciar el método respectivo que se encarga de ejecutar la solicitud y conocer además los objetos capaces de ejecutar la misma acción e invocarlos en un orden determinado, lo cual genera un alto acoplamiento entre objetos
- **Solución:** El patrón de diseño Chain of Responsability presenta una solución más óptima para cubrir este requerimiento, pues propone preparar cada objeto para que sea capaz de manejar una solicitud por su parte o a su vez de pasarle la solicitud a otro objeto definido dinámicamente en tiempo de ejecución, lo cual genera una cadena de objetos que son potenciales ejecutores de una solicitud, en caso de no poder ejecutar la solicitud, el objeto la pasa al siguiente eslabón de la cadena.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

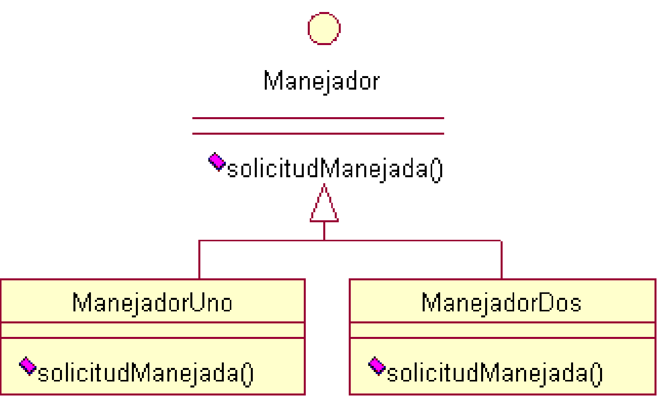


Figura 3.13: Patrón Chain of Responsability

A continuación se explica cada elemento de la Figura 3.13:

Manejador: Define una interfaz para tratar las peticiones.

Manejador Concreto: Trata las peticiones de las que es responsable, si no puede manejar la petición la reenvía a su sucesor.

APLICACIÓN:

Se recomienda el uso del patrón Chain of responsability cuando:

- Varios objetos deben manejar una solicitud, y determinar quien lo haga no es prioritario pues el objeto que la maneje es determinado automáticamente.
- Se desea realizar una solicitud a uno de entre varios objetos sin especificar explícitamente quien debe recibir la solicitud.
- El conjunto de objetos capaces de manejar una solicitud deben definirse dinámicamente.

COLABORACIONES:

- El patrón chain of responsability generalmente es implementado junto con el patrón composite, pues un componente padre puede ser un sucesor.

1.3.2. Command

DESCRIPCIÓN:

- **Objetivo:** Encapsular una solicitud en un objeto facilitando la parametrización de las clases clientes con diferentes consultas.
- **Problema:** Para la interacción entre objetos se manejan referencias directas entre los mismos, un objeto solicitante referencia los métodos de uno receptor; por medio de sentencias condicionantes (if o switch, case), se incrementa la complejidad cuando se trata de agregar una nueva clase receptora; ya que se debe modificar la clase invocadora aumentando la sentencia condicionante que discrimina la nueva clase; lo cual contradice los dos principios de la programación orientada a objetos[17].
- **Solución:** El patrón de diseño command soluciona esta contradicción mediante la creación de una clase abstracta llamada comando, la cual puede ser instanciada en una clase concreta, permitiendo parametrizar un objeto comando a ejecutar para la clase cliente.
- **Campo de acción:** aplicado a nivel de objetos.
- **Diagrama:**

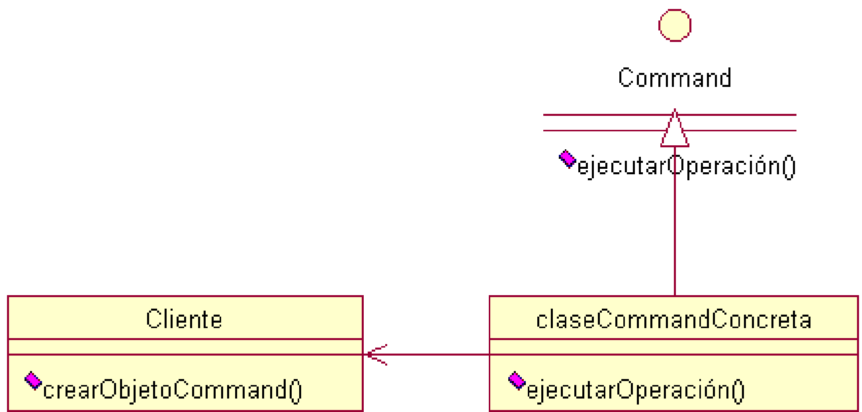


Figura 3.14: Patrón Command

A continuación se explica cada elemento de la Figura 3.14:

Command: declara la interface que ejecuta una operación.

[17] Abierto a la expansión y cerrado a la modificación, (Software Architecture Design Patterns in Java)

Clase Command Concreta: implementa la ejecución de la operación correspondiente.

Cliente: crea el objeto CommandConcreto.

APLICACIÓN:

Se recomienda el uso del patrón Command cuando:

- Se tienen comandos que diversos receptores pueden manejar de diferentes maneras.
- Se dispone de un conjunto de comandos de alto nivel que son implementados mediante operaciones primitivas.
- Se desea especificar, apilar y ejecutar comandos a diversos instantes.
- Es necesario disponer de la opción de deshacer la ejecución de comandos.
- Es necesario manejar auditorias y registros de todos los cambios mediante comandos.

COLABORACIONES:

- Un patrón composite puede ser usado para implementar comandos compuestos o transacciones, el uso del patrón memento conserva el estado que un comando requiere para poder deshacer los efectos de su ejecución. El uso del patrón prototype permite copiar un comando antes de ser grabado en un registro o histórico de acciones.

1.3.3. Interpreter

DESCRIPCIÓN:

- **Objetivo:** Dado un lenguaje de programación, define una representación con un intérprete que es usado para las sentencias.
- **Problema:** Para ejecutar diversos comandos o instrucciones, estas son definidas en una estructura básica para que la aplicación las interprete de manera correcta;

mediante un objeto que interpreta las sentencias por medio de condicionantes if o Switch, case: lo cual complica considerablemente la programación.

- **Solución:** El patrón de diseño Interpreter define a las instrucciones como objetos, lo cual permite que cada uno de estos que se implemente se identifique a sí mismo como una sentencia o a su vez a otra mediante un análisis de estructuras de árbol; todo este proceso mediante una interfaz.
- **Campo de acción:** aplicado a nivel de clases.
- **Diagrama:**

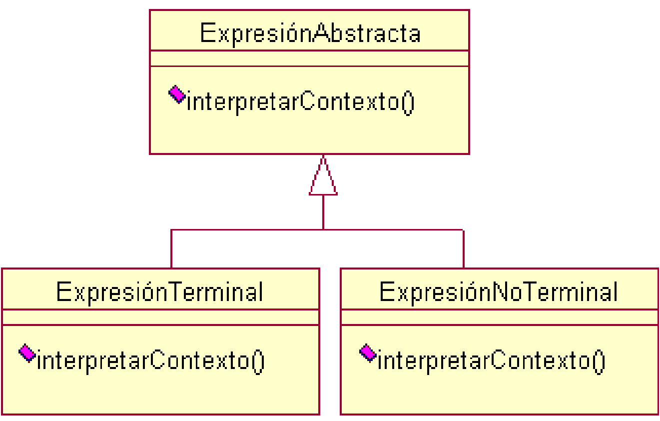


Figura 3.15: Patrón Interpreter

A continuación se explica cada elemento de la Figura 3.15:

Expresión Abstracta: declara las operaciones de interpretación que las especializaciones han de implementar.

Expresión Terminal: representa los símbolos terminales.

Expresión No Terminal: representa a las producciones de la gramática

APLICACIÓN:

Se recomienda el uso del patrón Interpreter cuando:

- Se desea interpretar una gramática, y esta no es muy compleja.
- La eficiencia esperada de la aplicación no es exigente.

COLABORACIONES:

- Las sentencias con estructura de árbol generalmente se implementan como un composite.
- El patrón flyweight muestra como compartir símbolos terminales dentro de la sintaxis de árbol abstracta.
- Se puede utilizar un patrón Iterator para recorrer la estructura.
- El patrón visitor puede ser utilizado para mantener el comportamiento de cada nodo en la estructura de árbol.

1.3.4. Iterator

DESCRIPCIÓN:

- **Objetivo:** Proveer una forma de acceder secuencialmente a los elementos de un objeto agregado, sin exponer su representación subyacente.
- **Problema:** Para manejar diversos objetos contenidos en una colección se debe la debe instanciar con su algoritmo de navegación; lo cual implica la creación de varios objetos, garantizando que la clase cliente conozca las implementaciones de los mismos, sobrecargando la memoria y provocando un alto acoplamiento entre los objetos y las clases que los instancian.
- **Solución:** El patrón de diseño Iterator implementa una interfaz para navegar a través de la colección de objetos, permitiendo que cada uno discrimine la secuencia a seguir; es decir que basta con instanciar la interfaz y cada objeto la implementará indicando el objeto actual y el siguiente, de acuerdo a algún criterio de ordenamiento de los elementos de la colección.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

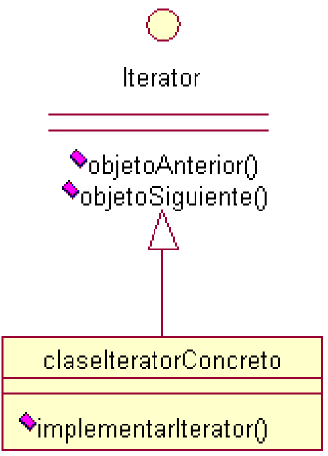


Figura 3.16: Patrón Iterator

A continuación se explica cada elemento de la Figura 3.16:

Iterator: declara la interface que permite acceder y navegar a través de los objetos.

Clase Iterator Concreto: implementa la interface Iterator.

APLICACIÓN:

Se recomienda el uso del patrón Iterator cuando:

- Se requiere manejar una colección de objetos y existen diferentes formas de navegar a través de ella.
- Existen diferentes colecciones de objetos para la misma lógica de navegación.
- Se pueden aplicar diferentes filtros y algoritmos de ordenamiento.

COLABORACIONES:

- Los patrones Iterator generalmente se aplican a estructuras recursivas, por lo que el uso del patrón composite es muy común.
- El patrón factory method permite instanciar el iterador apropiado para una colección.
- Generalmente se utiliza el patrón memento para conservar internamente el estado de cada iteración.

1.3.5. Mediator

DESCRIPCIÓN:

- **Objetivo:** Definir un objeto que encapsule la interacción entre otros, promoviendo un bajo acoplamiento; ya que evita que los objetos se referencien explícitamente entre ellos variando independientemente de su interacción.
- **Problema:** Todas las aplicaciones orientadas a objetos basan su funcionamiento en la interacción entre estos, si no existe un mecanismo claramente definido de interacción se puede generar una aplicación con un alto acoplamiento contradiciendo los principios de la programación orientada a objetos.
- **Solución:** El patrón de diseño Mediator, coordina interacciones entre objetos relacionados; centralizando en una clase la lógica que realiza los cambios de estados de los objetos, de manera que ofrece una forma sistematizada de aumentar la cohesión (se centraliza la lógica) y reducir el acoplamiento entre clases (se reducen la dependencias entre ellas).
- **Campo de acción:** aplicado a nivel de objetos.
- **Diagrama:**

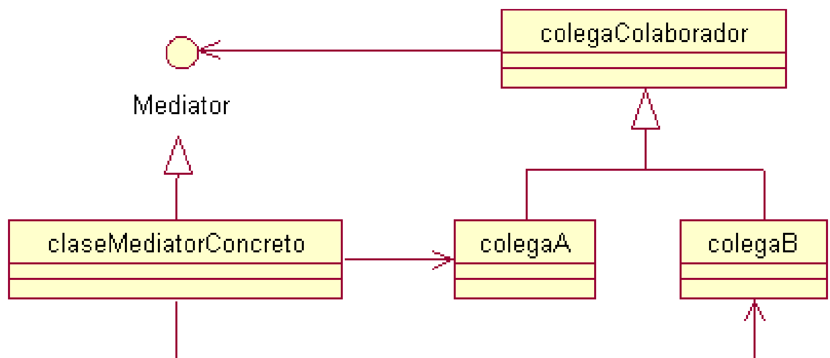


Figura 3.17: Patrón Mediator

A continuación se explica cada elemento de la Figura 3.17:

Mediator: declara la interface que permite comunicarse con los objetos colegas.

Colega Colaborador: permite comunicarse con la clase mediator y con otros colegas.

Clase Mediator Concreto: implementa la cooperación y colaboración entre los objetos colegas.

APLICACIÓN:

Se recomienda el uso del patrón Mediator cuando:

- Los objetos de la aplicación se comunican de manera bien estructura pero potencialmente complejas.
- Las identidades de los objetos deben protegerse aun cuando estos se comuniquen entre sí.
- El comportamiento de algunos objetos puede ser agrupado y personalizado.
- La reutilización de un objeto es complicada porque este referencia y se comunica con muchos otros objetos.

COLABORACIONES:

- Las clases Mediator frecuentemente utilizan el patrón de diseño Observer para recibir notificaciones de las diferentes solicitudes de las clases que interactúan.
- Se puede utilizar el patrón Adapter para independizar la clase Mediator de las clases concretas que gestiona.

1.3.6. Memento

DESCRIPCIÓN:

- **Objetivo:** Almacenar el estado interno de un objeto en un entorno externo al mismo, de manera que dicho estado pueda ser restaurado posteriormente.
- **Problema:** Una aplicación desea implementar la funcionalidad de deshacer acciones realizadas sobre objetos, lo que implica guardar sus estados; lo cual requiere que se implemente la funcionalidad de guardar y recuperar el estado en el objeto mismo, saturando la memoria.
- **Solución:** El patrón de diseño Memento sugiere separar al objeto de las funciones de guardar y recuperar un estado determinado, pero conservando al

mismo tiempo el encapsulamiento de los valores de los atributos del objeto, mediante la restricción de acceso de otros objetos a la persistencia de los estados del objeto.

- **Campo de acción:** aplicado a nivel de objetos.
- **Diagrama:**

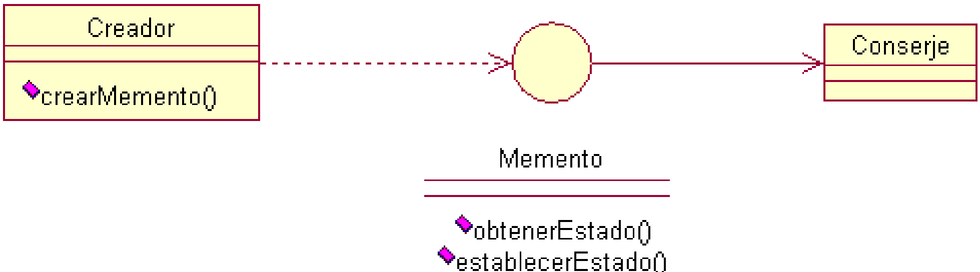


Figura 3.18: Patrón Memento

A continuación se explica cada elemento de la Figura 3.18:

Memento: se encarga de guardar tanta información, como sea necesario, del objeto Creador, especialmente su estado.

Originador: crea un objeto memento que contiene una instancia de su estado interno actual y emplea el objeto Memento para regresar a su estado anterior.

Conserje: almacena de manera segura al objeto memento.

APLICACIÓN:

Se recomienda el uso del patrón Memento cuando:

- Es necesario conservar el estado de un objeto a lo largo del tiempo (persistencia), y al mismo tiempo encapsular los valores de los atributos de dicho estado del objeto a objetos externos.

COLABORACIONES:

- Desconocida

1.3.7. Observer O Dependientes

DESCRIPCIÓN:

- **Objetivo:** Definir una o más dependencias entre objetos de tal manera que si un objeto cambia su estado, dichas relaciones sean notificadas y actualizadas automáticamente.
- **Problema:** La aplicación requiere que se cree dependencias entre objetos que permitan actualizar las relaciones entre los mismos en el momento en que uno varíe, en el desarrollo convencional este proceso implica código redundante y consumo amplio de recursos.
- **Solución:** El patrón Observer describe cómo establecer estas relaciones; empleando dos objetos esenciales: "sujeto" y "observador" que realizan el proceso conocido como publicar-subscribir; es decir sujeto puede manejar cualquier número de observadores dependientes, los que son notificados mediante una "publicación" cuando un sujeto sufre algún tipo de cambio en su estado, en ese momento observador le consultará para sincronizar su estado con el de el sujeto.
- **Campo de acción:** aplicado a nivel de objetos.
- **Diagrama:**

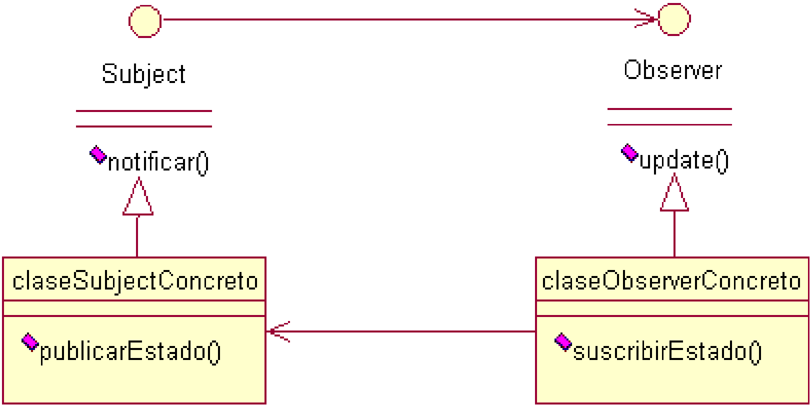


Figura 3.19: Patrón Observer

A continuación se explica cada elemento de la Figura 3.19:

Subject: provee una interfaz para detectar y acceder a los objetos Observadores.

Observer: define una interfaz para actualizar los objetos que deben ser notificados de los cambios de los sujetos.

Clase Subject Concreto: almacena el estado de los objetos Observer Concretos.

Clase Observer Concreto: mantiene una referencia a los objetos Subject Concretos.

APLICACIÓN:

Se recomienda el uso del patrón Observer cuando:

- Una abstracción requiere dos aspectos, uno dependiente del otro; los cuales necesitan ser encapsulados en los objetos separados para que pueden variar y ser usados de manera independiente.
- El cambio de un objeto requiere el cambio de otros, y no se conoce con exactitud cuántos.
- Un objeto debe notificar a otros sin necesidad de conocer quiénes son estos de manera específica.

COLABORACIONES:

- El patrón Mediator puede actuar como mediador entre los sujetos y los observadores, encapsulando la semántica de actualización compleja.
- El patrón Observer puede hacer uso del patrón Singleton para hacer único y globalmente accesible al sujeto.

1.3.8. State

DESCRIPCIÓN:

- **Objetivo:** Permitir a un objeto alterar su conducta cuando sus estados internos cambian; aparentemente parecerá que cambia sus clases.
- **Problema:** La aplicación requiere que se controle en tiempo de ejecución la variación de un objeto monolítico; dado que este es definido en función de su estado, si este varía el objeto también debe hacerlo; lo que implica crear una

clase de declaraciones dentro de la cual se determina que conducta se debe llevar a cabo, empleando un proceso poco práctico.

- **Solución:** El patrón State permite tener una clase que varía considerando las numerosas clases relacionadas; es decir que el patrón cambia entre las diferentes clases internas de tal manera que el objeto adjuntado parece cambiar de clase. Cuando se presentan declaraciones condicionantes el patrón coloca cada rama del condicionante en una clase separada tratando el estado del objeto como variable independiente de otros.
- Campo de acción: aplicado a nivel de objetos.
- **Diagrama:**

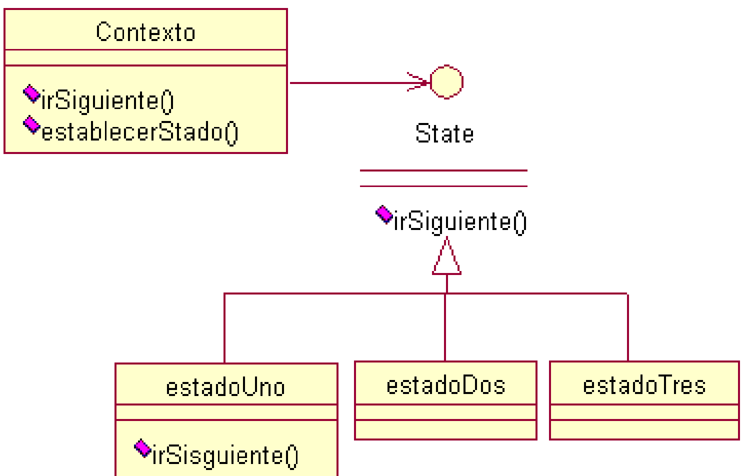


Figura 3.20: Patrón State

A continuación se explica cada elemento de la Figura 3.20:

Contexto: mantiene una instancia que define el estado actual del objeto.

State: define una interfaz para el encapsulamiento de la responsabilidad asociada con un estado particular de la clase contexto.

Estado "N": implementa el comportamiento o responsabilidad de la clase Contexto.

APLICACIÓN:

Se recomienda el uso del patrón State cuando:

- La conducta de un objeto depende totalmente de su estado, la cual debe cambiar en tiempo de ejecución de acuerdo a dicho estado.

- Las operaciones a ser realizadas tienen grandes declaraciones con muchas partes condicionantes que dependen del estado del objeto; el cual normalmente es representado por una o varias constantes numeradas

COLABORACIONES:

- El patrón Flyweight colabora en la implementación del patrón State; ya que explica cuando y donde deben ser compartidos los objetos States.

1.3.9. Strategy

DESCRIPCIÓN:

- **Objetivo:** Definir un grupo de algoritmos, encapsulando cada uno y haciéndolo intercambiable; esto permite que el algoritmo varíe independientemente de las clases clientes que lo usen.
- **Problema:** La aplicación requiere que se emplee uno de los principales conceptos de diseño “abrir-cerrar”; para alcanzar este objetivo se encapsula los detalles de las interfaces; pero esto no cubre los aspectos de cambios de clases y el impacto de la implementación de clases derivadas.
- **Solución:** El patrón Strategy crea un conjunto de algoritmos encapsulados y relacionados entre sí en una clase conductora llamada “Context”; el programa cliente puede seleccionar uno de los algoritmos o en algunos casos Context se encarga de seleccionar el más idóneo de acuerdo a la situación. El punto clave es que permite cambiar fácilmente entre algoritmos.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

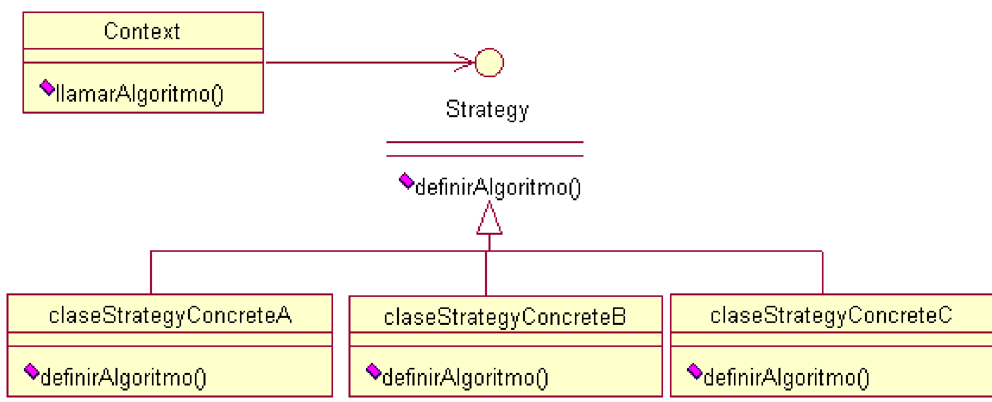


Figura 3.21: Patrón Strategy

A continuación se explica cada elemento de la Figura 3.21:

Strategy: declara una interfaz común para dar soporte a todos los algoritmos.

Context: mantiene una referencia al objeto Strategy.

Clase Strategy Concreta: implementa el algoritmo usando la interface Strategy.

APLICACIÓN:

Se recomienda el uso del patrón Strategy cuando:

- Varias clases relacionadas únicamente difieren en su conducta; ya que las estrategias proporcionan una manera de configurar una clase con una de muchas conductas.
- La aplicación necesita diferentes variantes del algoritmo; es decir que se podría definir un algoritmo que refleje los intercambios de una variable.
- Se requiere que un algoritmo use datos que los clientes no deben conocer; ya que el patrón evita exponer las estructuras complejas de los datos.
- Una clase debe definir muchas conductas; las cuales aparecen como múltiples declaraciones condicionantes en sus funcionamientos y el patrón se encarga de relacionarlas en ramas en su propia clase.

COLABORACIONES:

- El patrón Strategy eventualmente puede desarrollar adecuados objetos Flyweight.

1.3.10. Template

DESCRIPCIÓN:

- **Objetivo:** Definir la estructura de un algoritmo en una operación permitiendo a las subclases redefinir ciertos pasos sin cambiar la estructura del mismo.
- **Problema:** Es necesario emplear un mayor esfuerzo para poder proporcionar una interface común a componentes que han llegado a tener similitudes significantes; ya que implicaría duplicar el esfuerzo para poder detallar al máximo la relación entre dichos componentes.
- **Solución:** El patrón Template escribe una clase padre en la que se asignan uno o más métodos a ser implementados a las clases derivadas, formalizando la idea de definir un algoritmo en una clase; pero delegando los detalles de la implementación a las subclases.
- **Campo de acción:** aplicado a nivel de clases.
- **Diagrama:**

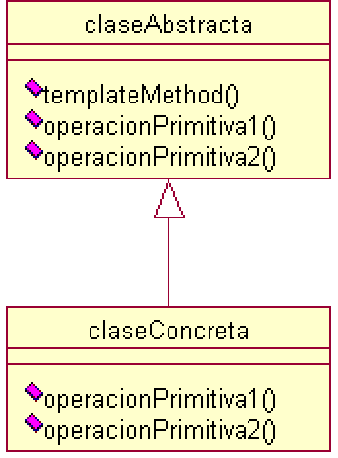


Figura 3.22: Patrón Template

A continuación se explica cada elemento de la Figura 3.22:

Clase Abstracta: define operaciones primitivas abstractas que cada subclase concreta define implementar.

Clase Concreta: implementa las operaciones primitivas.

APLICACIÓN:

Se recomienda el uso del patrón Template cuando:

- Se requiere implementar la parte invariable de un algoritmo en la clase y delegar las partes variables a las subclases.
- Una conducta común entre subclases debe factorizarse y localizarse en una clase común evitando código duplicado.
- La aplicación necesita controlar las extensiones de las subclases.

COLABORACIONES:

- El patrón Template presenta un funcionamiento semejante al patrón Strategy; usando la herencia para variar la parte de un algoritmo.

1.3.11. Visitor

DESCRIPCIÓN:

- **Objetivo:** Representar una operación a ser realizada en los elementos de una estructura de objetos; permitiendo definir un nuevo funcionamiento sin cambiar las clases de los elementos que operan.
- **Problema:** La aplicación requiere que se realicen muchas operaciones distintas que no se relacionen, en el objeto nodo de una estructura heterogénea agregada; tomando en cuenta que no se puede permitir llenar la clase nodo con este tipo de operaciones; incrementando la complejidad al analizar cada nodo hasta encontrar el adecuado para lanzar el indicador para realizar el funcionamiento deseado.
- **Solución:** El patrón Visitor es el modelo orientado a objetos que permite la creación de una clase externa que actúa sobre los datos de las otras clases, este proceso es altamente útil; ya que facilita el empleo únicamente de las instancias de las clases que se necesita para solventar las necesidades del momento sin necesidad de involucrar o afectar a las que no son útiles en determinada operación.
- **Campo de acción:** aplicado a nivel de objetos.

- **Diagrama:**

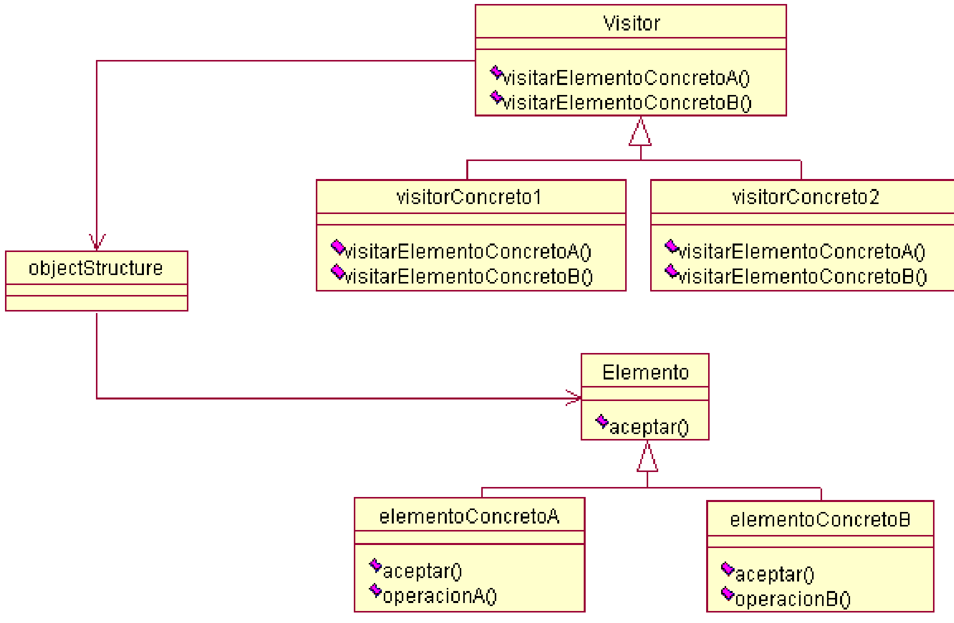


Figura 3.23: Patrón Visitor

A continuación se explica cada elemento de la Figura 3.23:

Visitor: declara una operación para cada clase en la estructura del objeto.

Visitor Concreto: implementa cada operación declarada en la clase Visitor.

Elemento: define una operación Aceptar que toma un Visitor como argumento.

Elemento Concreto: implementa cada la operación Aceptar.

ObjetStructure: provee una interfaz de alto nivel para permitir a Visitor visitar a sus elementos.

APLICACIÓN:

Se recomienda el uso del patrón Visitor cuando:

- Una estructura de objetos contiene muchas clases de objetos que difieren en su interface y el objetivo es combinar su funcionamiento de sus clases concretas.
- Muchas operaciones distintas y no relacionadas necesitan ser ejecutadas en objetos y estructuras de objetos; el objeto Visitor permite mantener unido el funcionamiento relacionado definiéndolo en una clase.

- Las clases definen estructuras de objetos que rara vez cambian; pero se necesita definir requerimientos altamente cambiantes.

COLABORACIONES:

- El patrón Visitor puede ser usado para aplicar una operación sobre una estructura de objetos definida por un patrón Composite.
- Los objetos Visitor pueden ser aplicados para la interpretación de objetos Interpreter.

2. Caracterización del sistema

Permite al equipo de desarrollo comprender de manera detallada el funcionamiento del sistema, con el fin de orientar las etapas de desarrollo a la optimización de recursos disponibles y al cumplimiento de las expectativas del cliente; para lo cual se definen a continuación los siguientes aspectos básicos:

- *Objetivo del sistema*
- *Requerimientos del sistema*

2.1. Objetivo del sistema

El equipo de desarrollo debe comprender claramente el objetivo principal del sistema, teniendo presente lo que el cliente espera, las tareas que deberá ejecutar, etc.

Aun cuando el objetivo principal del sistema es generalmente expresado por el cliente solicitante del desarrollo del mismo, es recomendable que el equipo de desarrollo lo estructure de la siguiente manera:

<verbo> <sujeto> <objeto> <complemento>

Donde:

Verbo es la acción a ejecutar en beneficio del sujeto, sujeto es la entidad beneficiada del cumplimiento del objetivo, objeto describe el medio piloto para beneficiar al objeto y complemento es la descripción detallada de la tarea a ejecutar para beneficiar al sujeto.

Así por ejemplo, un objetivo principal válido sería:

"Proveer al personal de la biblioteca una herramienta para la gestión automatizada de la biblioteca"

Verbo: Proveer.

Sujeto: Personal de la biblioteca.

Objeto: una herramienta.

Complemento: Gestión automatizada de la biblioteca.

Esta estructura del objetivo permite comprender claramente la necesidad principal de la aplicación, de manera que se pueda posteriormente priorizar funcionalidades de la misma.

2.2. Requerimientos del sistema

Los requerimientos del sistema son las necesidades que el cliente espera solventar mediante el uso de la aplicación.

Es recomendable que los requerimientos se expresen de forma técnica orientándolos a la fácil comprensión de los mismos por parte del equipo de desarrollo; para la documentación de especificación de requerimientos en un sistema orientado al uso de patrones de diseño es necesario incluir los pasos descritos en las siguientes secciones.

2.2.1. Requerimientos Funcionales

Define la funcionalidad que debe expresamente contemplar o no el sistema.

2.2.1.1. Responsabilidades

Definen todas las operaciones que el cliente espera cumpla el sistema, los requerimientos deberán describir además el flujo de los procesos, necesidades de almacenamiento, datos críticos y formulas de cálculo para procesos especiales.

Los requerimientos funcionales permiten al equipo de desarrollo abstraer los diferentes elementos necesarios para la construcción de la aplicación, de manera que se pueda identificar fácilmente capas, objetos, etc.

2.2.1.2. Exclusiones

Especifican detalles de funcionalidad que no le compete al sistema, son responsabilidades que están fuera del alcance de la aplicación, y que no deben ser consideradas ni en el diseño ni en la implementación de la aplicación.

2.2.2. Requerimientos no Funcionales

Los requerimientos no funcionales del sistema son aquellos que definen principalmente la *"Forma"* de la aplicación, es decir, aquellas necesidades que debe cumplir el sistema para cubrir las expectativas del cliente, pero que no se relacionan con procesos propios del negocio, sino mas bien con formalidades del sistema; por ejemplo tiempos de respuesta, plataforma de desarrollo, aplicaciones utilizadas para el desarrollo, incluso detalles de interfaces, colores, fuentes, etc.

3. Modelado de la solución

Se debe tener presente que los patrones de diseño son tecnologías de desarrollo aplicadas en programación orientada a objetos, por lo cual se recomienda para el modelado de la solución el uso de lenguajes de modelado, como UML; el uso de estos lenguajes permite simplificar el diseño de la aplicación desde la perspectiva del negocio, de tal manera que se obtenga un diseño del sistema especificando su estructura y comportamiento, resaltando las partes esenciales independientemente del lenguaje de programación a utilizarse en la implementación.

3.1. Creación de los casos de uso

Los casos de uso permiten describir entornos de operación de los diferentes actores del sistema, usuarios, clientes, sistemas externos, etc.

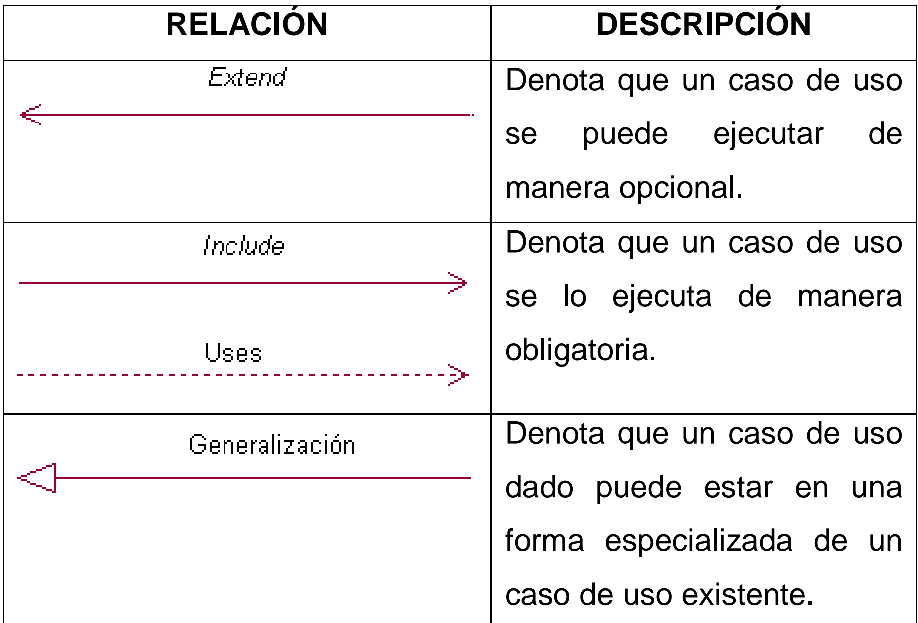
Es recomendable que los casos de uso se orienten a determinar que acciones se ejecutan en el entorno del negocio para el que va a ser desarrollada la aplicación, el objetivo de esta idea es que se puede definir varias tareas del sistema que cumplan con la actividad principal definida mediante el caso de uso. Los componentes básicos de un diagrama de casos de uso son los descritos en la Tabla 3.1.

NOMBRE	FIGURA	DESCRIPCIÓN
Actor		Entidad que utiliza uno de los casos de uso del sistema.
Caso de Uso		Denota una acción ejecutada a partir de una regla del negocio.

Tabla 3.1: Descripción Actores y Casos de uso[18]

[18] http://www.scribd.com/doc/6819691/Rose-creating-Usecase-and-Classdiagram

Además se debe identificar el tipo de relación que existe entre casos de uso, como se muestra en la Figura 3.2.



RELACIÓN	DESCRIPCIÓN
Extend	Denota que un caso de uso se puede ejecutar de manera opcional.
Include Uses	Denota que un caso de uso se lo ejecuta de manera obligatoria.
Generalización	Denota que un caso de uso dado puede estar en una forma especializada de un caso de uso existente.

Tabla 3.2: Descripción de relaciones entre Casos de uso[19]

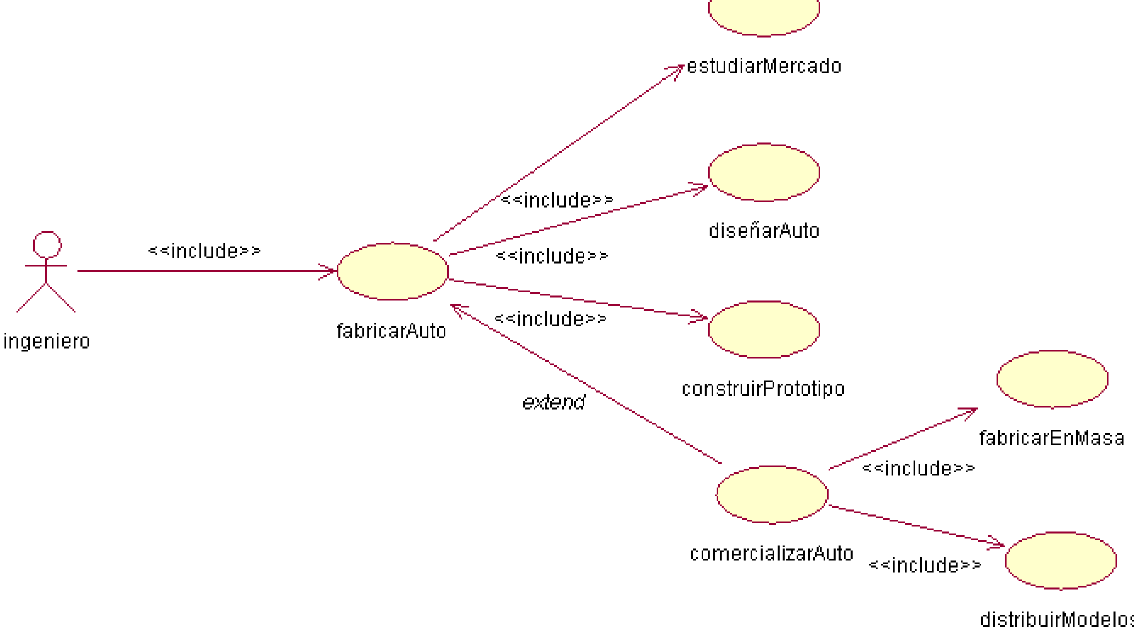


Figura 3.24: Ejemplo Diagrama de Casos de Uso

---

[19] http://www.scribd.com/doc/6819691/Rose-creating-Usecase-and-Classdiagram

4. Análisis detallado

El análisis detallado comprende el estudio minucioso de los requerimientos funcionales y los casos de uso; permite definir cada requerimiento funcional dentro del sistema.

4.1. Identificación de funcionalidades

Los casos de uso permiten identificar las funcionalidades requeridas de la aplicación; es decir, conocer qué procesos debe llevar a cabo la aplicación para cubrir sus requerimientos.

Cada requerimiento del sistema consiste en una tarea que el cliente desea automatizar o mejorar, y el diseño e implementación de dicha tarea conlleva implícitamente una serie de "retos" de ingeniería de software que seguramente ya fueron solventados por alguien más. Desde esta perspectiva, se puede decir que los patrones de diseño constituyen el catálogo de las soluciones a problemas (retos) de diseño e implementación del software.

4.1.1. Identificación de componentes

Identificar los elementos que deberán interactuar entre sí para lograr completar el funcionamiento del sistema, en programación orientada a objetos, los elementos de un sistema se implementan como clases, y en tiempo de ejecución se denominan objetos.

Para identificar los objetos de la aplicación, es recomendable utilizar la "abstracción", es decir representar los objetos del mundo real mediante sus dos elementos principales: características y comportamiento; la abstracción toma estas

características y las representa mediante atributos mientras que el comportamiento lo representa mediante funciones que cada objeto debe ser capaz de ejecutar.

4.2. Identificación de problemas

Tras conocer los casos de uso y realizar un análisis detallado del funcionamiento interno de la aplicación, se puede fácilmente predecir las principales complicaciones de diseño que surgirán a lo largo del ciclo de vida de la solución.

Debido a la clasificación de los patrones de diseño, se recomienda agrupar los posibles problemas acorde a las categorías de los mismos. Las categorías de los problemas solucionados por los patrones de diseño son:

- **Problemas de creación de objetos (Instanciación)**
  - Alto acoplamiento entre objetos, al cambiar el nombre de un objeto, se deberá reflejar este cambio en cada una de las clases que instancian dicho objeto.
  - La creación descontrolada de objetos genera una sobrecarga de memoria.
  - La creación de objetos puede complicarse si ésta requiere tomar decisiones.

Los problemas de creación de objetos inciden directamente sobre la escalabilidad de la aplicación, así como también sobre los tiempos de respuesta y el mantenimiento de la misma.

- **Problemas de comportamiento de objetos**
  - La redundancia de código, por operaciones similares realizadas por diversos objetos.

- La interacción entre objetos puede llegar a complicar altamente la estructura del sistema cuando ésta no es controlada apropiadamente.

Los problemas de comportamiento de objetos inciden directamente sobre la complejidad de la aplicación y, por lo tanto, sobre el entendimiento de la misma por parte de recursos técnicos que puedan estar involucrados en mejoras y mantenimiento de la misma.

- **Problemas de la estructura de los objetos**
  - Las relaciones entre objetos permite generar estructuras más complejas, si estas relaciones no se controlan, la complejidad del sistema puede aumentar descontroladamente.
  - El uso de clases estáticas imposibilita la recomposición de objetos en tiempo de ejecución, lo cual limita el alcance de nueva funcionalidad.

Los problemas de la estructura de objetos inciden sobre la complejidad de la aplicación, los recursos usados por la misma en tiempo de ejecución, limitando su escalabilidad y mantenimiento.

4.2.1. Selección de patrones candidatos

Considerando que una meta del desarrollo de software consiste en simplificar el diseño, es importante tener en cuenta que los patrones de diseño son soluciones a problemas puntuales, por lo que se los debe incluir en el desarrollo de software solo en caso de que se presenten problemas que sean resueltos por esos patrones. Una mala práctica de desarrollo es intentar agregar patrones de diseño por simplemente agregarlos, en este caso se puede complicar demasiado el diseño de una solución.

Para facilitar el uso de los patrones a partir de los problemas descritos en la sección 4.1, esta guía presenta los patrones de diseño agrupados de acuerdo a las categorías más comunes de las soluciones que presentan (ver Tabla 3.4)

4.2.2. Análisis del catálogo de patrones de diseño

Tras la identificación de los patrones candidatos, se debe leer el detalle de los patrones de diseño en el catálogo (sección 1); de tal manera que se pueda tener una idea clara de la aplicabilidad de cada patrón y poder optar por la decisión que se considere la más apropiada para el desarrollo de la aplicación.

4.3. *Diseño del diagrama de clases de la aplicación*

Para la creación del diagrama de clases se deben tener claro los objetos, funcionalidades y las relaciones entre objetos que permitirán a la aplicación cubrir sus requerimientos funcionales.

Se debe tener presente que una clase es capaz de englobar las características generales de un grupo de objetos, por lo que se puede entender que una clase es en realidad un grupo de objetos, para implementar un diagrama de clase se cuenta con los elementos definidos en la Tabla 3.3.

NOMBRE	FIGURA	DESCRIPCIÓN
Clase	Nueva Clase1 atributo1 operacion1()	Una clase es un contenedor de uno o más atributos (variables o propiedades) junto a las operaciones de manipulación de dichos datos (funciones o métodos). De manera general es la definición de un objeto o un grupo de objetos.
Variables	atributo1	Son características de los objetos representadas por un nombre y su tipo, conocidas también como el almacén de datos de un estado relacionado con el objeto.

Método	operacion1()	Es la funcionalidad asociada al objeto, que permiten definir el comportamiento del mismo para que este pueda interactuar con otros elementos de la aplicación.
Propiedad	setAtributo() getAtributo()	Son un tipo especial de métodos que permiten consultar o modificar el valor de variables miembro privadas; cuyo acceso es controlado.

Tabla 3.3: Descripción Clases[20]

[20] http://www.scribd.com/doc/6819691/Rose-creating-Usecase-and-Classdiagram

OBJETIVO	CREACIONALES	ESTRUCTURALES	COMPORTAMIENTO
Reducir el acoplamiento entre objetos	• Abstract Factory • Builder • Factory Method	• Facade	• Chain of Responsibility • Iterator • Mediator
Implementar soluciones controlando la complejidad	• Builder • Factory Method • Prototype	• Adapter • Composite	• Command • Interpreter • State • Template • Visitor
Facilitar el mantenimiento posterior de la aplicación	• Factory Method (Re-ingeniería) • Todos	• Todos	• Todos
Mantener un bajo consumo de recursos en tiempo de ejecución	• Prototype • Singleton	• Flyweight • Proxy	• Iterator • Memento • Observer
Mejorar el control sobre un objeto	• Singleton	-----------	-----------
Permitir la reutilización de soluciones	• Todos	• Bridge (Re ingeniería) • Todos	• Todos
Mejorar la escalabilidad de la aplicación	• Todos	• Bridge (Re ingeniería) • Facade (Re ingeniería) • Todos	• Strategy (Re ingeniería) • Todos
Proveer dinamismo a la aplicación en tiempo de ejecución	• Builder • Factory Method	• Decorator	• Todos
Omitir la redundancia de código	-----------	-----------	• Observer

Tabla 3.4: Clasificación de las soluciones presentadas por los patrones de diseño

Además se debe identificar el tipo de relación que existe entre las clases y los objetos definidos para el sistema.

RELACIÓN	DESCRIPCIÓN
Generalización Especialización	Denota que una subclase hereda los métodos y atributos especificados por una Súper Clase.
Asociación	Denota la relación de asociación objetos que colaboran entre sí.
<<Dependencia>>	Denota una relación en la que una clase es instanciada y la misma depende de otro objeto o clase.

Tabla 3.5: Descripción relaciones entre clases[21]

En programación orientada a objetos, cada componente del sistema se implementa mediante clases, y cada vez que en tiempo de ejecución se instancie una de estas clases, se obtendrá un objeto.

4.3.1. Elaborar el diagrama de clases

Desarrollar el diagrama de clases incluyendo los métodos y relaciones entre clases; además identificar y representar las que serán afectadas por un patrón de diseño; como se muestra en la figura 3.25.

[21] http://www.scribd.com/doc/6819691/Rose-creating-Usecase-and-Classdiagram

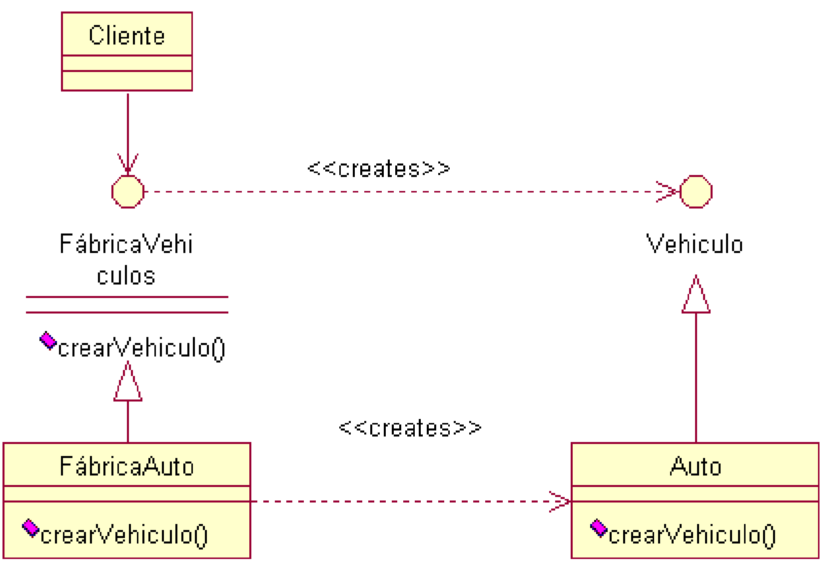


Figura 3.25: Ejemplo Diagrama de Clases

4.3.2. Definición de patrones participantes

Otorgar un nombre a los patrones participantes; de tal manera que puedan ser fácilmente identificados entre las clases con las que interactuarán.

5. Arquitectura de la solución

5.1. Definición de las capas de la aplicación

Capa de presentación: conocida también como "Capa de usuario"; es la encargada de presentar el sistema al usuario, le provee información y captura información del usuario en un mínimo de proceso. Es de gran utilidad porque realiza un filtrado previo para comprobar que no hay errores de formato. Esta capa se comunica únicamente con la capa de negocio.

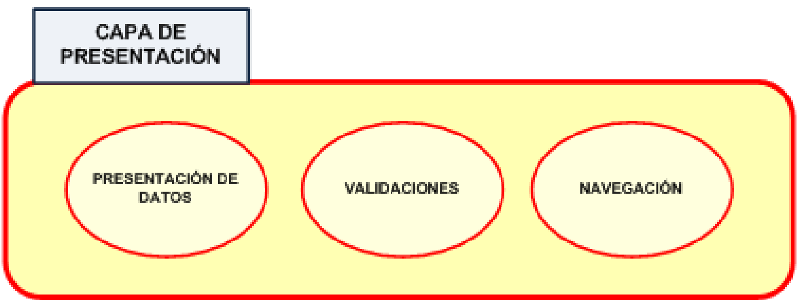


Figura 3.26: Capa de Presentación

Capa de negocio: conocida también como "Lógica del negocio" es donde residen las reglas que deben cumplirse; recibe las solicitudes y entrega los resultados.

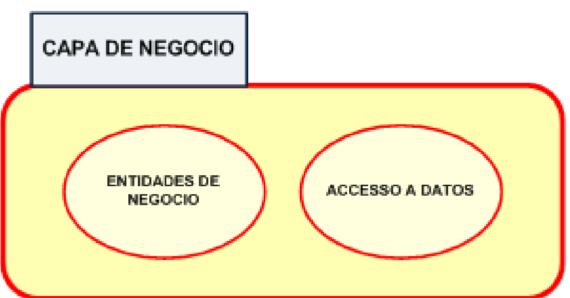


Figura 3.27: Capa de Negocio

Capa de datos: conocida también como "Capa de almacenamiento" es la encargada de guardar los datos; puede ser formada con uno o más gestores de base de datos mismos que se encargan del almacenamiento o recuperación.

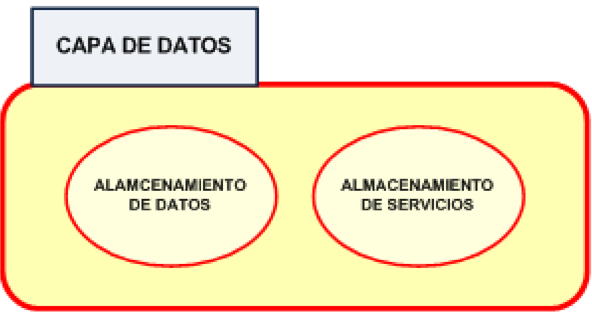


Figura 3.28: Patrón Builder

5.2. Elaboración del diagrama de despliegue

En el diagrama de despliegue se deben ubicar los patrones de acuerdo a su aporte para cada capa de la aplicación, mimas que se encuentran ilustradas en la figura 3.29 y cuentan con los siguientes patrones de diseño:

Capa de datos:

- Singleton
- Proxy
- Command
- Adapter
- Flywieght
- Interpreter

Capa de Negocio:

- Composite
- Mediator
- Bridge
- Command
- Adapter
- Memento
- Chain of Responsability
- Observer
- Strategy
- Factory Method
- Abstract Factory
- Interpreter

Capa de presentación:

- Decorator
- Builder
- Command

- Interpreter
- Adapter
- Facade

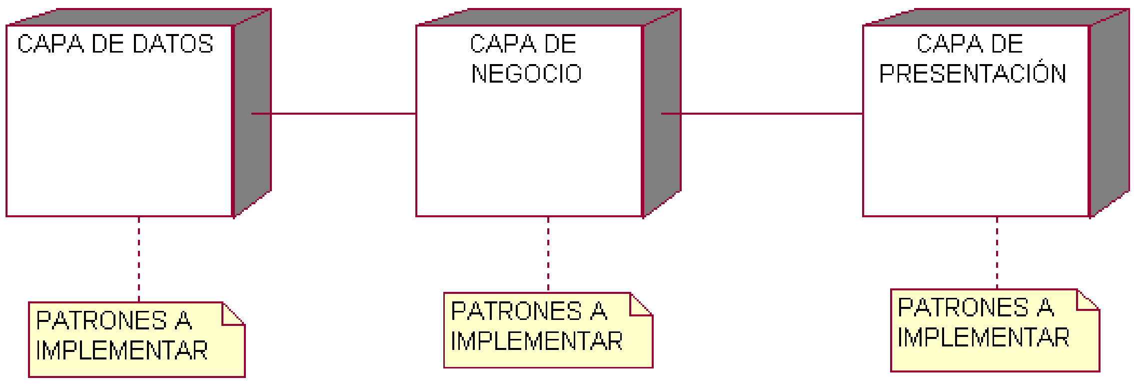


Figura 3.29: Diagrama de Despliegue

6. Diseño de la solución

6.1. Diseño del modelo Entidad-Relación

Una de las principales ventajas de modelar empleando UML es que provee un diagrama de clases que permite definir claramente la solución, mismo que es la base para realizar el modelo Entidad-Relación; el cual es de mucha importancia ya que representará a las tablas de bases de datos participantes con sus respectivos atributos.

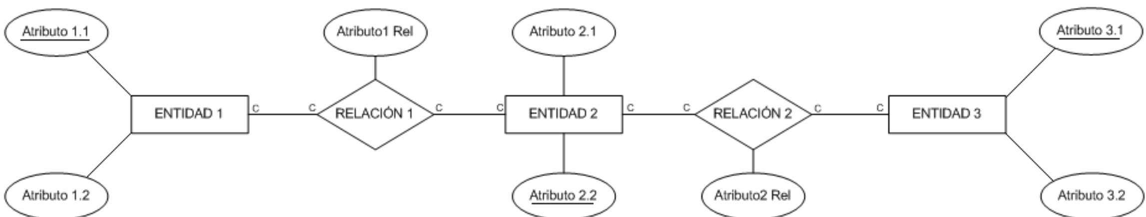


Figura 3.30: Modelo Entidad-Relación

6.2. *Diseño del modelo conceptual*

6.2.1. Definición de las tablas

Otorgar un nombre a cada una de las tablas, representadas por entidades en el diagrama Entidad-Relación; además considerar a los atributos para los cuales se definirá un tipo de dato.

Tabla: Entidad 1

Atributos	Tipo Dato	Restricción
ATRIBUTO_1.1		
ATRIBUTO_1.2		

Tabla 3.6: Ejemplo de una tabla

1.1.1. Elaboración del modelo

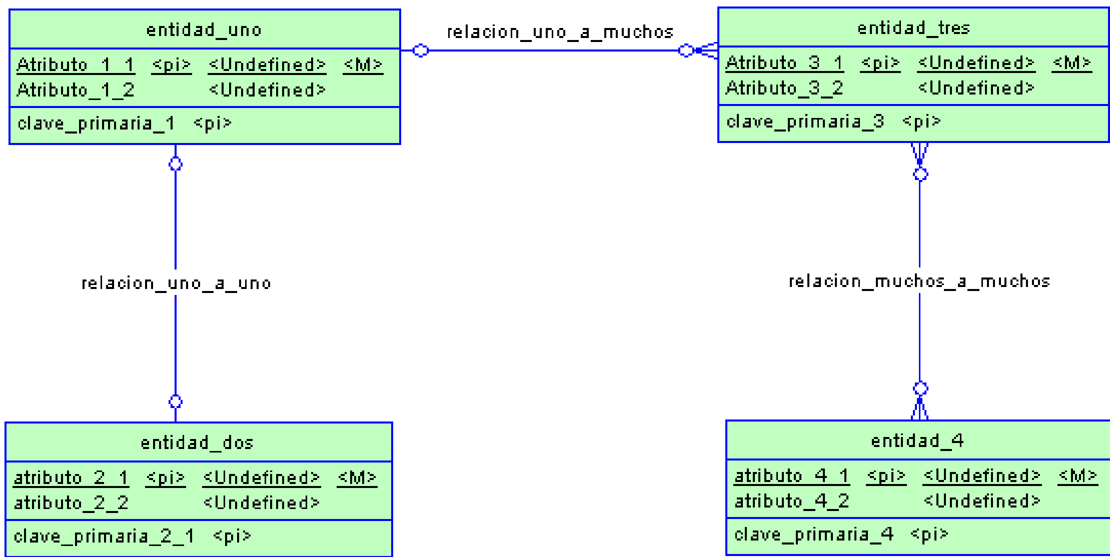


Figura 3.31: Modelo Conceptual

6.3. Diseño del modelo físico

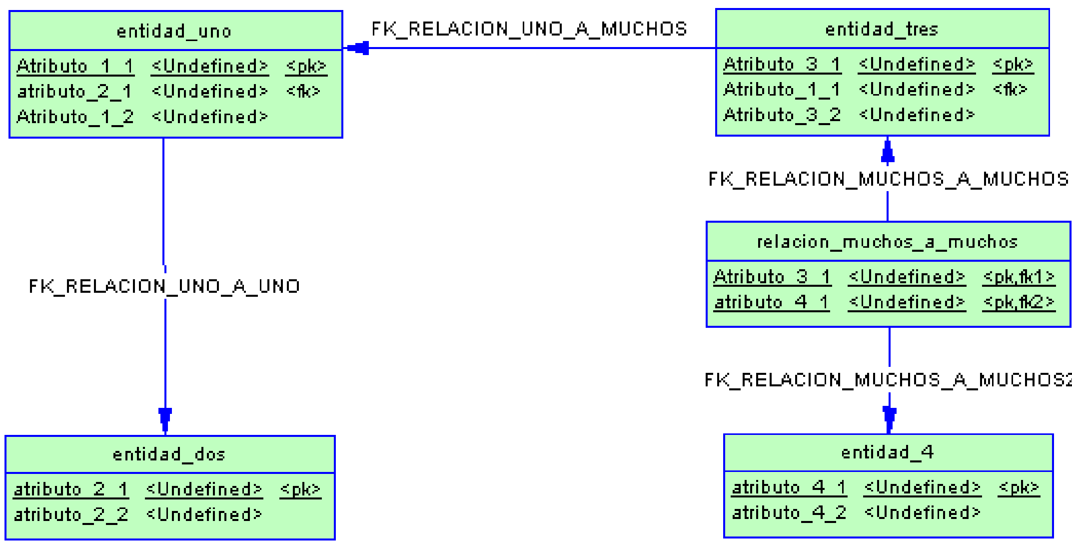


Figura 3.32: Modelo Físico

7. Codificación del sistema

Finalmente se deberá codificar (programar) la aplicación, acorde a los diseños de la misma.

7.1. Creación de la base de datos de la aplicación

Basándose en el modelo físico de datos se deberá generar el script de creación de la base de datos para la aplicación, así como también se deberán crear un usuario de bdd y los permisos del mismo para poder utilizarlo como usuario de datos de la aplicación.

7.2. Implementación de las capas definidas en la arquitectura de la solución

Basándose en los diagramas de la aplicación, se deberán codificar las clases, de acuerdo a cada capa definida para la aplicación; es muy importante tener presente

qué patrones de diseño van a intervenir en el proceso de codificación de manera sencilla y práctica como se puede observar en el Ejemplo 3.1.

Ejemplo 3.1: implementación del patrón de diseño "Abstract Factory" para la construcción de vehículos.

```
public interfase FabricaVehiculos
{
 CrearVehiculo();
}

public class FabricaAuto: FabricaVehiculos
{
  public Vehiculo CrearVehiculo()
  {
    return new Vehiculo();
  }
}
```

Figura 3.33: Codificación patrón Abstract Factory

CAPÍTULO IV

4. APLICACIÓN DE LA GUÍA EN UN CASO DE ESTUDIO

En el presente capítulo se describe la aplicación de patrones de diseño al desarrollo de software en un sistema que servirá para automatizar la *gestión de una biblioteca.*

4.1. DESCRIPCIÓN DEL CASO DE ESTUDIO

Para el caso de estudio se va a aplicar RUP, pues este proceso de desarrollo es 100% orientado a objetos y presenta gran flexibilidad a la hora de implementarlo, lo cual permite optimizar los tiempos invertidos en el desarrollo de una aplicación.

A lo largo del capítulo, a manera de ejemplo, se presentarán extractos de modelos y diagramas del *módulo de seguridades* del sistema.

4.1.1. IDENTIFICACIÓN DEL CASO DE ESTUDIO

La selección del caso de estudio partió de la necesidad de desarrollar un sistema con las suficientes prestaciones para poder aplicar ampliamente patrones de diseño, y de esta manera retroalimentar las actividades descritas en la guía, al aplicarla en un caso real de desarrollo de software.

El caso de estudio a desarrollar debe manejar una conexión a una base de datos, de manera que pueda demostrarse de mejor manera el uso de patrones de diseño para controles de seguridad, adicionalmente, el caso de estudio debe corresponder a una aplicación orientada a objetos.

4.2. APLICACIÓN DE LA GUÍA EN LA IMPLEMENTACIÓN DEL CASO DE ESTUDIO

A continuación se implementarán paso a paso las actividades descritas en la guía para el uso de patrones de diseño en el desarrollo de software, estas actividades son:

- Caracterización del sistema
- Modelado de la solución
- Análisis detallado
- Arquitectura de la solución
- Diseño de la aplicación
- Codificación del sistema
- Análisis del uso de la guía en el caso de estudio

4.2.1. CARACTERIZACIÓN DEL SISTEMA

La caracterización del sistema constituye n proceso de entendimiento de la solución por parte del equipo de desarrollo. Para el caso de estudio se ha optado por desarrollar un sistema que automatice la gestión de una biblioteca.

4.2.1.1. Objetivo

El sistema debe ser capaz de agilitar la gestión de los libros existentes en una biblioteca.

4.2.1.2. Requerimientos del Sistema

A continuación se describirán los requisitos de desarrollo del sistema; cuales funciones deberá cumplir y cuales funciones no están consideradas para su desarrollo.

4.2.1.2.1. Requerimientos Funcionales

Aquellos requerimientos que definirán la funcionalidad del sistema:

4.2.1.2.1.1. Responsabilidades

- Registrar, modificar, eliminar y buscar registros de las publicaciones disponibles en la biblioteca, con las siguientes consideraciones:
  - Registro: La creación de publicaciones debe incluir la información correspondiente a:
    - Titulo
    - Autor (es)
    - Edición
    - Tomo
    - Materia (Política, literatura, ciencias, ingeniería, etc.)
    - Año de publicación
    - Editorial
    - Tipo de publicación (Libro, revista, Artículo, etc.)
    - Formato de publicación (Impreso, digital, página web, etc.)
    - Estado (Disponible, agotado, en revisión, etc.)
  - Las búsquedas pueden ser por cualquiera de los atributos de las publicaciones.
- Crear, modificar, eliminar y buscar información correspondiente a las suscripciones de usuarios de la biblioteca, con las siguientes consideraciones.
  - Creación: La creación de un suscriptor deberá contener la información personal de un cliente, mas referencias que permitirán categorizar al cliente para determinar costos de suscripción y plazos de préstamos.
- Manejar un registro de préstamos de publicaciones realizados por la biblioteca con las siguientes consideraciones:

- La creación de un registro de préstamos deberá controlar el stock de publicaciones, y que el cliente no tenga ninguna limitación por mora o incumplimiento de cualquier tipo con la biblioteca.
- Automáticamente la aplicación deberá notificar al usuario cuando un cliente esté a punto de exceder la fecha límite de entrega de una publicación, así como considerar aquellos préstamos que hayan superado ya su fecha de entrega para inhabilitar nuevos préstamos a ese cliente y calcular el valor de la multa a ser cancelada por dicho incumplimiento.

- Manejar los registros mediante una base de datos.

4.2.1.2.1.2. Exclusiones

- El sistema no manejará información contable de ningún tipo, considerando que el costo de suscripción y el valor de la multa son datos informativos para el personal de la biblioteca.
- El sistema no se encargará de administrar los registros del personal.
- El sistema no se encargará de la facturación de los valores de suscripción cancelados por los clientes.
- El sistema no manejará información correspondiente a proveedores de publicaciones de ningún tipo (donantes, distribuidoras, etc.).

4.2.1.2.2. No Funcionales

Los siguientes requisitos no funcionales serán implementados para cubrir requerimientos complementarios del usuario.

- Seguridad: Control de acceso a la aplicación por usuarios y perfiles de usuario.
- Mantenibilidad: La aplicación deberá ser capaz de soportar cambios a lo largo de su vida útil.
- Confiabilidad: La aplicación no deberá presentar errores en su ejecución.

4.2.2. MODELADO DE LA SOLUCIÓN

El sistema estará compuesto por los siguientes módulos:

- *Módulo de gestión de publicaciones*
- *Módulo de gestión de clientes*
- *Módulo de gestión de préstamos*
- *Módulo de seguridad*

4.2.2.1. CASOS DE USO

a) Módulo de gestión de publicaciones

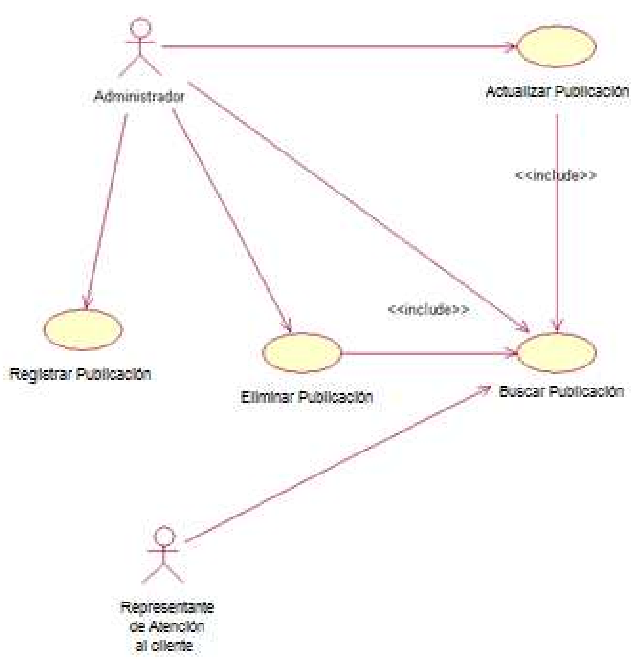


Figura 4.1: Diagrama de caso de uso para el requerimiento funcional 1.

b) Módulo de gestión de clientes

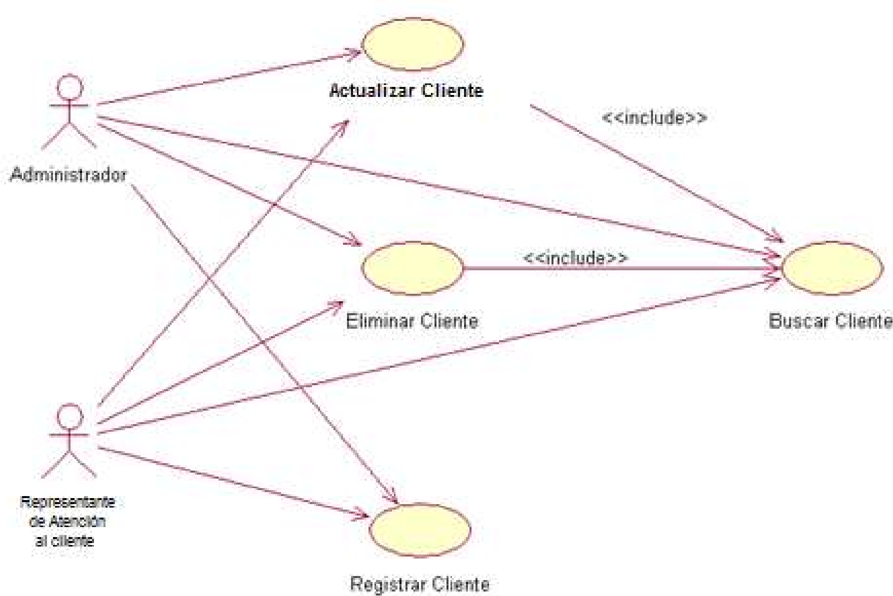


Figura 4.2: Diagrama de caso de uso para el requerimiento funcional 2

c) Módulo de gestión de Préstamos

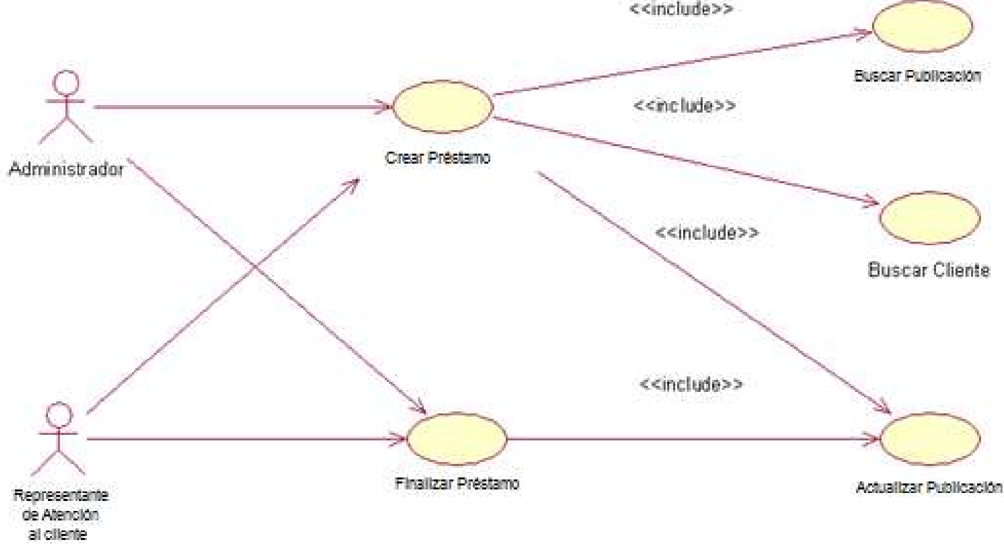


Figura 4.3: Diagrama de caso de uso para el requerimiento funcional 3

4.2.3. ANÁLISIS DETALLADO

En esta sección se ilustrará la solución a desarrollarse en un entorno técnico.

4.2.3.1. FUNCIONALIDADES

El sistema debe:

- Gestionar publicaciones: Crear, modificar, almacenar, buscar y eliminar registros de publicaciones.
- Gestionar atributos de publicaciones: Crear, modificar, almacenar, buscar, eliminar y vincular atributos con publicaciones.
- Gestionar clientes: Crear, modificar, almacenar, buscar y eliminar registros de clientes de la biblioteca.
- Gestionar las categorías de clientes: Crear, modificar, almacenar, buscar, eliminar y vincular categorías con clientes.
- Gestionar los préstamos de publicaciones a clientes:
  - Crear un préstamo vinculando la publicación prestada con el cliente solicitante del préstamo, considerando el stock de la publicación y la factibilidad de acceso a préstamos de publicaciones por parte del cliente.
  - Eliminar el registro de un préstamo por devolución de la publicación por parte del cliente, se deberá además actualizar el stock de la publicación y la disponibilidad del cliente para acceder a préstamos
  - Notificar al usuario del sistema cuando un cliente haya incumplido o vaya a incumplir con los plazos de préstamos de una publicación, considerando que se debe actualizar la disponibilidad del cliente a acceder a nuevos préstamos.
- Gestionar usuarios del sistema: Crear, modificar, almacenar, buscar y eliminar registros de usuarios del sistema.
- Gestionar perfiles de usuario: Crear, modificar, almacenar, buscar, eliminar y vincular perfiles con usuarios.

- Gestionar permisos por perfiles de usuario: Crear, modificar, almacenar, buscar, eliminar y vincular permisos con perfiles de usuario.

4.2.3.1.1. Identificación de componentes

Los elementos que el sistema debe contener son:

- Elementos de abstracción del negocio
  - Publicaciones
  - Clientes
  - Usuarios
  - Perfiles del usuario
  - Préstamos
- Elementos de abstracción complementarios
  - Atributos de publicaciones
  - Permisos de perfil de usuarios
- Elementos de funcionalidad
  - Gestores de acceso a datos
  - Gestores de menús por perfil
  - Gestores de Interfaces de uso del sistema (GUI's)
  - Gestores de objetos de abstracción
  - Gestores de reglas del negocio

4.2.3.2. IDENTIFICACIÓN DE PROBLEMAS

Los posibles problemas en el desarrollo de la aplicación son:

- **Problemas de creación de objetos (Instanciación)**
  - Complejidad en la creación de objetos, debido a la necesidad de dinamizar la aplicación y su funcionamiento.
  - Limitada escalabilidad y mantenimiento complejo de la aplicación en caso de que se acoplen los objetos creadores con los objetos creados.

- o Sobrecarga de memoria en la creación de objetos similares (publicaciones)
- o Falta de control en la creación y acceso a objetos.

- **Problemas de la estructura de los objetos**
  - o Complejidad de desarrollo, escalabilidad y mantenimiento de la aplicación debido al uso de objetos compuestos por uno o varios objetos jerárquicos, como perfiles de usuario y publicaciones.

4.2.3.2.1. Selección de patrones candidatos

A partir de los problemas presentes en el sistema, se consideran los siguientes patrones:

- Builder y abstract factory para controlar la complejidad en la creación de objetos.
- Singleton para controlar el acceso a objetos.
- Composite para controlar la complejidad de la aplicación por la gestión de objetos de estructura compleja.
- Command y mediator para separar a los objetos de las operaciones que los gestionan, mejora l mantenibilidad de la aplicación.

4.2.3.2.2. Análisis del catálogo de patrones

Tras analizar el catálogo de patrones de diseño, se eligieron los patrones builder, singleton, composite y command.

4.2.3.3. DIAGRAMA DE CLASES DE LA APLICACIÓN

Para esta sección se ha descrito la aplicación de la guía al módulo de seguridades, en el cual se puede apreciar claramente la inclusión de patrones de diseño, para observar toda la aplicación, referirse al Anexo 1.

4.2.3.3.1. Diagrama del Módulo de seguridades

Los patrones de diseño incluidos en el diagrama de clases mostrado en la Figura 4.4, son: Composite y Command, estos patrones se describen en la siguiente sección.

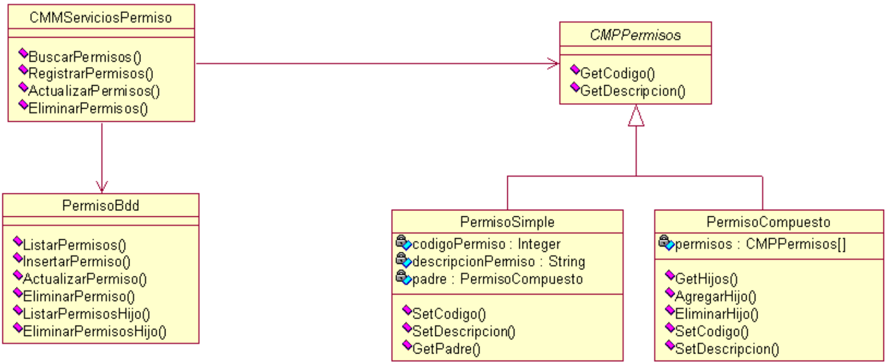


Figura 4.4: Implementación del requerimiento de gestión de permisos de perfiles de usuario

4.2.3.3.2. Definición de Patrones Participantes

En la Figura 4.4, podemos apreciar a los siguientes patrones de diseño:

CMPPermisos: de tipo composite, permite gestionar fácilmente la jerarquía de permisos sin necesidad de redundar código para gestionar elementos simples y compuestos, adicionalmente, en caso de que se desee incorporar nuevas características a un elemento de tipo composite, simplemente se agregarán estas características en las clases concretas (PermisoSimple y PermisoCompuesto), y al referenciarlos mediante su clase abstracta, esta actualización de características no implicará una actualización de las clases clientes del patrón.

CMMServiciosPermiso: de tipo command, permite separar la implementación de un objeto, de las operaciones que lo gestionan, sujetándonos al caso anterior, en caso de requerir nuevas operaciones de gestión de los objetos, se podrá actualizar este objeto, y hacer uso de la nueva funcionalidad, sin que los objetos que utilizaban la funcionalidad anterior se vean afectados.

4.2.4. ARQUITECTURA DE LA SOLUCIÓN

La arquitectura de la solución permite identificar claramente las capas que compondrán la aplicación.

4.2.4.1. DEFINICIÓN DE CAPAS DE LA APLICACIÓN

La definición de las capas de la aplicación permitirá agregar modularidad al sistema. Según la Figura 4.5, podemos clasificar la aplicación según las capas:

Capa de presentación: Builder
Capa de herramientas: Command
Capa de abstracción: Composite
Capa de acceso a datos: Singleton

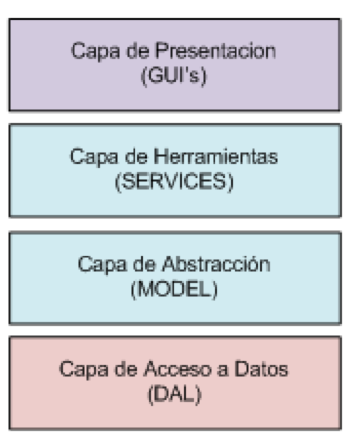


Figura 4.5: Diagrama de Arquitectura de la solución

4.2.4.2. DIAGRAMA DE DESPLIEGUE

El diagrama de despliegue de la Figura 4.6 muestra como interactuarán las diversas capas de la aplicación.

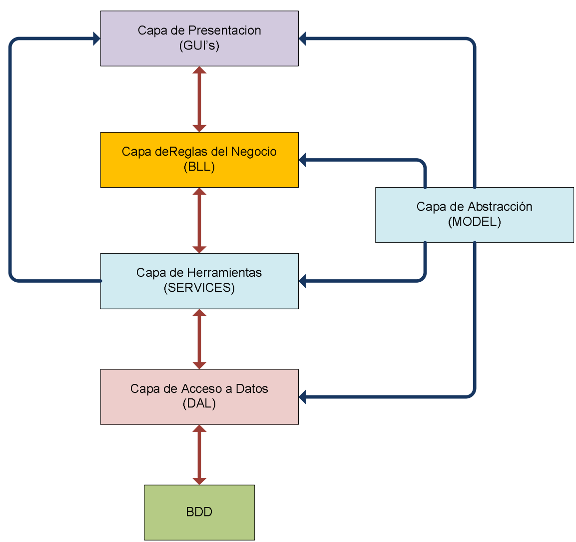


Figura 4.6: Diagrama de despliegue de la aplicación

4.2.5. DISEÑO DE LA SOLUCIÓN

En el diseño se van a incluir el modelo conceptual y modelo físico de la base de datos de la aplicación

4.2.5.1. MODELO CONCEPTUAL (REQUERIMIENTO DE CONTROL DE ACCESO POR PERFILES)

El diseño conceptual muestra las entidades de BDD que se van a utilizar para permitir a la aplicación gestionar las seguridades.

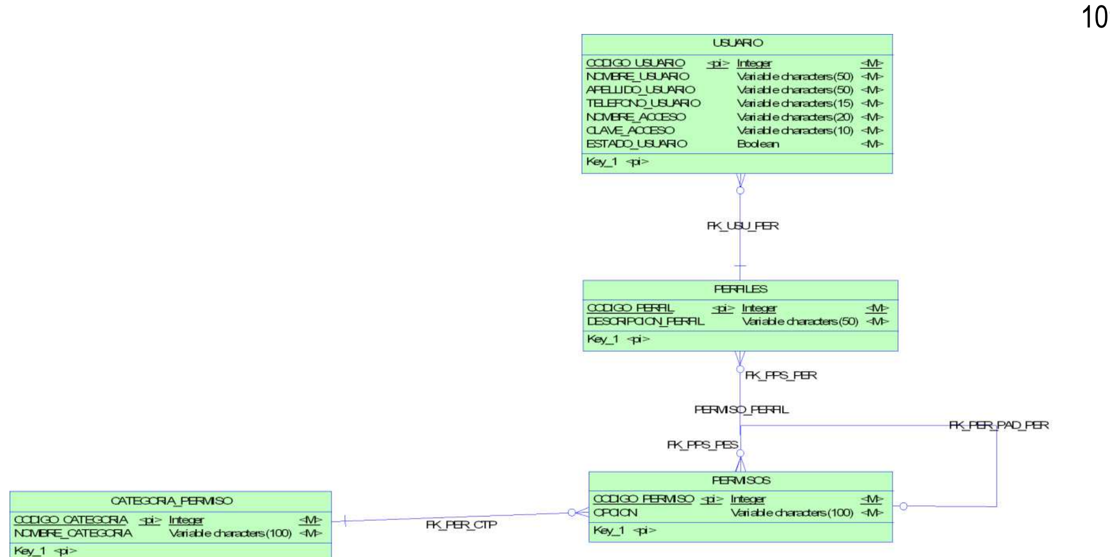


Figura 4.7: Modelo conceptual de la solución

4.2.5.2. MODELO FÍSICO (REQUERIMIENTO DE CONTROL DE ACCESO POR PERFILES)

El diseño físico muestra las tablas resultantes de la BDD, tras la implementación del modelo conceptual.

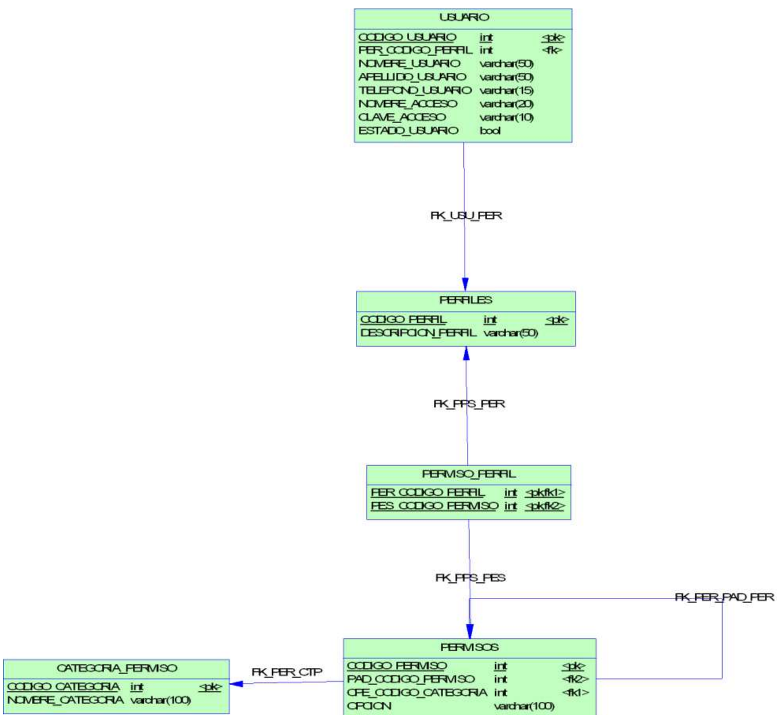


Figura 4.8: Modelo físico de la solución

4.2.6. CODIFICACIÓN DE SISTEMA

En esta sección, se incluirán los códigos de las clases que componen el módulo de gestión de seguridades.

4.2.6.1. CREACIÓN DE LA BDD DEL REQUERIMIENTO

La creación de las siguientes tablas permite identificar claramente la aplicación del patrón de diseño desde la base de datos, la sección resaltada permite identificar la relación de un elemento con otro de la misma de la tabla, el campo admite nulos, pues no todo elemento tendrá un elemento padre, en la Figura 4.9 se muestra la estructura.

```
/*==============================================================*/
/* Table: CATEGORIA_PERMISO                                     */
/*==============================================================*/
create table CATEGORIA_PERMISO
(
   CODIGO_CATEGORIA     int not null,
   NOMBRE_CATEGORIA     varchar(100) not null,
   primary key (CODIGO_CATEGORIA)
);

/*==============================================================*/
/* Table: PERMISO_PERFIL                                        */
/*==============================================================*/
create table PERMISO_PERFIL
(
   PER_CODIGO_PERFIL        int not null,
   PES_CODIGO_PERMISO       int not null,
   primary key (PER_CODIGO_PERFIL, PES_CODIGO_PERMISO)
);

/*==============================================================*/
/* Table: PERFILES                                              */
/*==============================================================*/
create table PERFILES
(
   CODIGO_PERFIL        int not null,
   DESCRIPCION_PERFIL   varchar(50) not null,
   primary key (CODIGO_PERFIL)
);

/*==============================================================*/
/* Table: PERMISOS                                              */
/*==============================================================*/
create table PERMISOS
(
   CODIGO_PERMISO       int not null,
   PAD_CODIGO_PERMISO   int,
   CPE_CODIGO_CATEGORIA     int not null,
   OPCION               varchar(100) not null,
   primary key (CODIGO_PERMISO)
);

/*==============================================================*/
/* Table: USUARIO                                               */
/*==============================================================*/
create table USUARIO
(
   CODIGO_USUARIO       int not null,
   PER_CODIGO_PERFIL        int not null,
   NOMBRE_USUARIO       varchar(50) not null,
   APELLIDO_USUARIO     varchar(50) not null,
   TELEFONO_USUARIO     varchar(15) not null,
   NOMBRE_ACCESO        varchar(20) not null,
   CLAVE_ACCESO         varchar(10) not null,
   ESTADO_USUARIO       bool not null,
   primary key (CODIGO_USUARIO)
);
alter table PERMISO_PERFIL add constraint FK_PPS_PER foreign key (PER_CODIGO_PERFIL)
      references PERFILES (CODIGO_PERFIL) on delete restrict on update restrict;

alter table PERMISO_PERFIL add constraint FK_PPS_PES foreign key (PES_CODIGO_PERMISO)
      references PERMISOS (CODIGO_PERMISO) on delete restrict on update restrict;

alter table PERMISOS add constraint FK_PER_CTP foreign key (CPE_CODIGO_CATEGORIA)
      references CATEGORIA_PERMISO (CODIGO_CATEGORIA) on delete restrict on update restrict;

alter table PERMISOS add constraint FK_PER_PAD_PER foreign key (PAD_CODIGO_PERMISO)
      references PERMISOS (CODIGO_PERMISO) on delete restrict on update restrict;

alter table USUARIO add constraint FK_USU_PER foreign key (PER_CODIGO_PERFIL)
      references PERFILES (CODIGO_PERFIL) on delete restrict on update restrict;
```

Figura 4.9: Tablas ejemplo del modelo de BDD

4.2.6.2. CODIFICACIÓN DEL REQUERIMIENTO DE CONTROL DE ACCESO POR PERFILES

Capa de Acceso de Datos: En esta capa se controlará el acceso a la BDD de la aplicación, toda conexión a la BDD se realizará a través de esta capa, de manera que se pueda asegurar la estabilidad de los datos. El código presentado en las Ilustraciones 4.1 a 4.5, corresponde a las clases de acceso a la BDD de la aplicación, en esta se gestionan las consultas SQL que permitirán persistir los datos de un objeto.

```
package biblioteca.bdd;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;
import biblioteca.dal.DalParameter;
import biblioteca.dal.ParameterType;
import biblioteca.model.Perfil;
public class PerfilBdd {
	private String SQL_SELECT_PERFIL = "SELECT codigo_perfil, " +
	   "descripcion_perfil " +
	   "FROM perfiles ";
	private String TX_INSERT_PERFIL = "INSERT INTO perfiles(" +
	  "codigo_perfil, " +
	  "descripcion_perfl) " +
	  "VALUES (?, ?)";
	private String TX_UPDATE_PERFIL = "UPDATE perfiles " +
	  "SET descripcion_perfil =? " +
	  "WHERE codigo_perfil = ?";
	private String TX_DELETE_PERFIL = "DELETE FROM perfiles " +
	  "WHERE codigo_perfil = ?";
	public ResultSet ListarPerfiles(Perfil perfilBuscado) throws SQLException
	{
		String sql = SQL_SELECT_PERFIL;
		boolean set = false;
		Collection<DalParameter> parms = new ArrayList<DalParameter>();
		int index = 1;
		if (perfilBuscado.getCodigoPerfil() > 0)
		{
			sql += "WHERE codigo_perfil LIKE CONCAT('%', ?, '%') ";
			DalParameter parm = new DalParameter();
			parm.setIndex(index);
			index++;
			parm.setType(ParameterType.INTEGER);
			parm.setValue(String.valueOf(perfilBuscado.getCodigoPerfil()));
			parms.add(parm);
			set = true;
		}
		if (!perfilBuscado.getDescripcionPerfil().isEmpty())
```

```
            {
                  if (set)
                        sql += "AND descripcion_perfil LIKE CONCAT('%', ?, '%')";
                  else
                        sql += "WHERE descripcion_perfil LIKE CONCAT('%', ?, '%')
";
                  DalParameter parm = new DalParameter();
                  parm.setIndex(index);
                  index++;
                  parm.setType(ParameterType.STRING);
                  parm.setValue(perfilBuscado.getDescripcionPerfil());
                  parms.add(parm);
            }
            PreparedStatement sta = Herramientas.getConn().prepareStatement(sql);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            ResultSet rs = sta.executeQuery();
            return rs;
      }
      public void InsertarPerfil(Perfil perfilAInsertar) throws SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter codigoPerfil = new DalParameter();
            codigoPerfil.setIndex(1);
            codigoPerfil.setType(ParameterType.INTEGER);
      codigoPerfil.setValue(String.valueOf(perfilAInsertar.getCodigoPerfil()));
            parms.add(codigoPerfil);
            DalParameter descripcionPerfil = new DalParameter();
            descripcionPerfil.setIndex(2);
            descripcionPerfil.setType(ParameterType.STRING);
      descripcionPerfil.setValue(String.valueOf(perfilAInsertar.getDescripcionPerfi
l()));
            parms.add(descripcionPerfil);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_INSERT_PERFIL);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
      public void ActualizarPerfil(Perfil perfilAActualizar) throws SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
```

```
            DalParameter descripcionPerfil = new DalParameter();
            descripcionPerfil.setIndex(1);
            descripcionPerfil.setType(ParameterType.STRING);
            descripcionPerfil.setValue(perfilAActualizar.getDescripcionPerfil());
            parms.add(descripcionPerfil);
            DalParameter codigoPerfil = new DalParameter();
            codigoPerfil.setIndex(2);
            codigoPerfil.setType(ParameterType.INTEGER);
      codigoPerfil.setValue(String.valueOf(perfilAActualizar.getCodigoPerfil()));
            parms.add(codigoPerfil);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_UPDATE_PERFIL);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
      public void EliminarPerfil(Perfil perfilAEliminar) throws SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter codigoPerfil = new DalParameter();
            codigoPerfil.setIndex(1);
            codigoPerfil.setType(ParameterType.INTEGER);
      codigoPerfil.setValue(String.valueOf(perfilAEliminar.getCodigoPerfil()));
            parms.add(codigoPerfil);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_DELETE_PERFIL);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
}
```

Ilustración 4.1: Código de la clase PerfilBdd, capa de acceso a datos para el objeto “Perfil”

```
package biblioteca.bdd;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;
import biblioteca.dal.DalParameter;
import biblioteca.dal.ParameterType;
import biblioteca.model.CMPPermiso;
```

```
import biblioteca.model.PermisoSimple;
import biblioteca.model.PermisoCompuesto;
public class PermisoBdd {
	private String SQL_SELECT_PERMISO = "SELECT codigo_permiso, " +
		"pad_codigo_permiso, " +
		"descripcion_permiso " +
		"FROM permisos ";
	private String TX_INSERT_PERMISO_HIJO = "INSERT INTO permisos(" +
		"codigo_permiso, " +
		"descripcion_permiso, " +
		"pad_codigo_permiso) " +
		"VALUES (?, ?, ?)";
	private String TX_INSERT_PERMISO = "INSERT INTO permisos(" +
		"codigo_permiso, " +
		"descripcion_permiso) " +
		"VALUES (?, ?)";
	private String TX_UPDATE_PERMISO_HIJO = "UPDATE permisos " +
		"SET descripcion_permiso =?, " +
		"pad_codigo_permiso =? " +
		"WHERE codigo_permiso = ?";
	private String TX_UPDATE_PERMISO = "UPDATE permisos " +
		"SET descripcion_permiso =?, " +
		"WHERE codigo_permiso = ?";
	private String TX_DELETE_PERMISO = "DELETE FROM permisos " +
		"WHERE codigo_permiso = ?";
	private String TX_DELETE_PERMISO_HIJO = "DELETE FROM permisos " +
		"WHERE pad_codigo_permiso = ?";
	public ResultSet ListarPermisos(CMPPermiso permisoBuscado) throws
SQLException
	{
		String sql = SQL_SELECT_PERMISO;
		boolean set = false;
		Collection<DalParameter> parms = new ArrayList<DalParameter>();
		int index = 1;
		if (permisoBuscado.getCodigoPermiso() > 0)
		{
			sql += "WHERE codigo_permiso LIKE CONCAT('%', ?, '%') ";
			DalParameter parm = new DalParameter();
			parm.setIndex(index);
			index++;
```

```
                  parm.setType(ParameterType.INTEGER);
      parm.setValue(String.valueOf(permisoBuscado.getCodigoPermiso()));
                  parms.add(parm);
                  set = true;
            }
            if (!permisoBuscado.getDescripcionPermiso().isEmpty())
            {
                  if (set)
                        sql += "AND descripcion_permiso LIKE CONCAT('%', ?, '%')
";
                  else
                        sql += "WHERE descripcion_permiso LIKE CONCAT('%', ?,
'%') ";
                  DalParameter parm = new DalParameter();
                  parm.setIndex(index);
                  index++;
                  parm.setType(ParameterType.STRING);
                  parm.setValue(permisoBuscado.getDescripcionPermiso());
                  parms.add(parm);
            }
            PreparedStatement sta = Herramientas.getConn().prepareStatement(sql);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            ResultSet rs = sta.executeQuery();
            return rs;
      }
      public ResultSet ListarPermisosRaiz(CMPPermiso permiso) throws SQLException
      {
            String sql = SQL_SELECT_PERMISO;
            Collection<DalParameter> parms = new ArrayList<DalParameter>();
            boolean set = false;
            int index = 1;
            if (permiso.getCodigoPermiso() > 0)
            {
                  sql += "WHERE codigo_permiso LIKE CONCAT('%', ?, '%') ";
                  DalParameter parm = new DalParameter();
                  parm.setIndex(index);
                  index++;
                  parm.setType(ParameterType.INTEGER);
                  parm.setValue(String.valueOf(permiso.getCodigoPermiso()));
                  parms.add(parm);
```

```
                    set = true;
                }
                if (set)
                    sql += "AND pad_codigo_permiso IS null ";
                else
                    sql += "WHERE pad_codigo_permiso IS null ";
                PreparedStatement sta = Herramientas.getConn().prepareStatement(sql);
                sta = Herramientas.getPreparedStatementReady(sta, parms);
                ResultSet rs = sta.executeQuery();
                return rs;
        }
        public ResultSet ListarPermisosHijo(PermisoCompuesto permisoPadre) throws
SQLException
        {
                String sql = SQL_SELECT_PERMISO;
                Collection<DalParameter> parms = new ArrayList<DalParameter>();
                int index = 1;
                if (permisoPadre.getCodigoPermiso() > 0)
                {
                    sql += "WHERE pad_codigo_permiso = ? ";
                    DalParameter parm = new DalParameter();
                    parm.setIndex(index);
                    index++;
                    parm.setType(ParameterType.INTEGER);
                    parm.setValue(String.valueOf(permisoPadre.getCodigoPermiso()));
                    parms.add(parm);
                }
                PreparedStatement sta = Herramientas.getConn().prepareStatement(sql);
                sta = Herramientas.getPreparedStatementReady(sta, parms);
                ResultSet rs = sta.executeQuery();
                return rs;
        }
        public void InsertarPermiso(CMPPermiso permisoAInsertar) throws SQLException
        {
                Collection<DalParameter> parms  = new ArrayList<DalParameter>();
                DalParameter codigoPermiso = new DalParameter();
                codigoPermiso.setIndex(1);
                codigoPermiso.setType(ParameterType.INTEGER);
        codigoPermiso.setValue(String.valueOf(permisoAInsertar.getCodigoPermiso()));
                parms.add(codigoPermiso);
```

```
            DalParameter descripcionPermiso = new DalParameter();
            descripcionPermiso.setIndex(2);
            descripcionPermiso.setType(ParameterType.STRING);
            descripcionPermiso.setValue(permisoAInsertar.getDescripcionPermiso());
            parms.add(descripcionPermiso);
            PreparedStatement sta;
            if
(permisoAInsertar.getClass().getName().equalsIgnoreCase(PermisoSimple.class.getName(
)))
            {
                  DalParameter codigoPermisoPadre = new DalParameter();
                  codigoPermisoPadre.setIndex(3);
                  codigoPermisoPadre.setType(ParameterType.INTEGER);
      codigoPermisoPadre.setValue(String.valueOf(((PermisoSimple)permisoAInsertar).
getPadre().getCodigoPermiso()));
                  parms.add(codigoPermisoPadre);
                  sta =
Herramientas.getConn().prepareStatement(TX_INSERT_PERMISO_HIJO);
            }
            else
                  sta =
Herramientas.getConn().prepareStatement(TX_INSERT_PERMISO);

            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
      public void ActualizarPermiso(CMPPermiso permisoAActualizar) throws
SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter descripcionPermiso = new DalParameter();
            descripcionPermiso.setIndex(1);
            descripcionPermiso.setType(ParameterType.STRING);
      descripcionPermiso.setValue(permisoAActualizar.getDescripcionPermiso());
            parms.add(descripcionPermiso);
            PreparedStatement sta;
            if
(permisoAActualizar.getClass().getName().equalsIgnoreCase(PermisoSimple.class.getNam
e()))
            {
```

```
                    DalParameter codigoPermisoPadre = new DalParameter();
                    codigoPermisoPadre.setIndex(2);
                    codigoPermisoPadre.setType(ParameterType.INTEGER);
      codigoPermisoPadre.setValue(String.valueOf(((PermisoSimple)permisoAActualizar
).getPadre().getCodigoPermiso()));
                    parms.add(codigoPermisoPadre);
                    DalParameter codigoAtributo = new DalParameter();
                    codigoAtributo.setIndex(3);
                    codigoAtributo.setType(ParameterType.INTEGER);
      codigoAtributo.setValue(String.valueOf(permisoAActualizar.getCodigoPermiso())
);
                    parms.add(codigoAtributo);
                    sta =
Herramientas.getConn().prepareStatement(TX_UPDATE_PERMISO_HIJO);
             }
             else
             {
                    DalParameter codigoAtributo = new DalParameter();
                    codigoAtributo.setIndex(2);
                    codigoAtributo.setType(ParameterType.INTEGER);
      codigoAtributo.setValue(String.valueOf(permisoAActualizar.getCodigoPermiso())
);
                    parms.add(codigoAtributo);
                    sta =
Herramientas.getConn().prepareStatement(TX_UPDATE_PERMISO);
             }
             sta = Herramientas.getPreparedStatementReady(sta, parms);
             sta.executeUpdate();
      }
      public void EliminarPermiso(CMPPermiso permisoAEliminar) throws SQLException
      {
             Collection<DalParameter> parms  = new ArrayList<DalParameter>();
             DalParameter codigoPermiso = new DalParameter();
             codigoPermiso.setIndex(1);
             codigoPermiso.setType(ParameterType.INTEGER);
      codigoPermiso.setValue(String.valueOf(permisoAEliminar.getCodigoPermiso()));
             parms.add(codigoPermiso);
             PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_DELETE_PERMISO);
             sta = Herramientas.getPreparedStatementReady(sta, parms);
```

```
            sta.executeUpdate();
      }
      public void EliminarPermisosHijo(CMPPermiso permisoPadre) throws SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter codigoPadre = new DalParameter();
            codigoPadre.setIndex(1);
            codigoPadre.setType(ParameterType.INTEGER);
            codigoPadre.setValue(String.valueOf(permisoPadre.getCodigoPermiso()));
            parms.add(codigoPadre);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_DELETE_PERMISO_HIJO);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
}
```

Ilustración 4.2: Código de la clase PermisoBdd, capa de acceso a datos para el objeto "Permiso"

```
package biblioteca.bdd;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;
import biblioteca.dal.DalParameter;
import biblioteca.dal.ParameterType;
import biblioteca.model.CategoriaPermiso;
public class CategoriaPermisoBdd {
      private String SQL_SELECT_CATEGORIA_PERMISO = "SELECT codigo_categoria, " +
            "nombre_categoria " +
            "FROM categoria_permiso ";
      private String TX_INSERT_CATEGORIA_PERMISO = "INSERT INTO categoria_permiso("
            + "codigo_categoria, " +
             "nombre_categoria) " +
             "VALUES (?, ?)";
      private String TX_UPDATE_CATEGORIA_PERMISO = "UPDATE categoria_permiso " +
             "SET nombre_categoria =? " +
             "WHERE codigo_categoria = ?";
      private String TX_DELETE_CATEGORIA_PERMISO = "DELETE FROM categoria_permiso "
            + "WHERE codigo_categoria = ?";
```

```
	public ResultSet ListarCategorias(CategoriaPermiso categoriaBuscada) throws
SQLException
	{
		String sql = SQL_SELECT_CATEGORIA_PERMISO;
		boolean set = false;
		Collection<DalParameter> parms = new ArrayList<DalParameter>();
		int index = 1;
		if (categoriaBuscada.getCodigoCategoria() > 0)
		{
			sql += "WHERE codigo_categoria LIKE CONCAT('%', ?, '%') ";
			DalParameter parm = new DalParameter();
			parm.setIndex(index);
			index++;
			parm.setType(ParameterType.INTEGER);
		parm.setValue(String.valueOf(categoriaBuscada.getCodigoCategoria()));
			parms.add(parm);
			set = true;
		}
		if (!categoriaBuscada.getNombreCategoria().isEmpty())
		{
			if (set)
				sql += "AND nombre_categoria LIKE CONCAT('%', ?, '%') ";
			else
				sql += "WHERE nombre_categoria LIKE CONCAT('%', ?, '%')
";
			DalParameter parm = new DalParameter();
			parm.setIndex(index);
			index++;
			parm.setType(ParameterType.STRING);
			parm.setValue(categoriaBuscada.getNombreCategoria());
			parms.add(parm);
		}
		PreparedStatement sta = Herramientas.getConn().prepareStatement(sql);
		sta = Herramientas.getPreparedStatementReady(sta, parms);
		ResultSet rs = sta.executeQuery();
		return rs;
	}
	public void InsertarCategoria(CategoriaPermiso categoriaAInsertar) throws
SQLException
	{
```

```
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter codigoCategoria = new DalParameter();
            codigoCategoria.setIndex(1);
            codigoCategoria.setType(ParameterType.INTEGER);
      codigoCategoria.setValue(String.valueOf(categoriaAInsertar.getCodigoCategoria
()));
            parms.add(codigoCategoria);
            DalParameter nombreCategoria = new DalParameter();
            nombreCategoria.setIndex(2);
            nombreCategoria.setType(ParameterType.STRING);
            nombreCategoria.setValue(categoriaAInsertar.getNombreCategoria());
            parms.add(nombreCategoria);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_INSERT_CATEGORIA_PERMISO);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
      public void ActualizarCategoria(CategoriaPermiso categoriaAActualizar) throws
SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter nombreCategoria = new DalParameter();
            nombreCategoria.setIndex(1);
            nombreCategoria.setType(ParameterType.STRING);
            nombreCategoria.setValue(categoriaAActualizar.getNombreCategoria());
            parms.add(nombreCategoria);
            DalParameter codigoCategoria = new DalParameter();
            codigoCategoria.setIndex(2);
            codigoCategoria.setType(ParameterType.INTEGER);
      codigoCategoria.setValue(String.valueOf(categoriaAActualizar.getCodigoCategor
ia()));
            parms.add(codigoCategoria);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_UPDATE_CATEGORIA_PERMISO);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
      public void EliminarCategoria(CategoriaPermiso categoriaAEliminar) throws
SQLException
      {
```

```
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter codigoCategoria = new DalParameter();
            codigoCategoria.setIndex(1);
            codigoCategoria.setType(ParameterType.INTEGER);
      codigoCategoria.setValue(String.valueOf(categoriaAEliminar.getCodigoCategoria
()));
            parms.add(codigoCategoria);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_DELETE_CATEGORIA_PERMISO);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
}
```

Ilustración 4.3: Código de la clase CategoríaPermisoBdd, capa de acceso a datos para el objeto "CategoriaPermiso"

```
package biblioteca.bdd;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;
import biblioteca.dal.DalParameter;
import biblioteca.dal.ParameterType;
import biblioteca.model.CMPPermiso;
import biblioteca.model.Perfil;
public class PermisoPerfilBdd {
      private String SQL_SELECT_PERMISO_PERFIL = "SELECT per_codigo_perfil, " +
         "pes_codigo_permiso " +
         "FROM permiso_perfil ";
      private String TX_INSERT_PERMISO_PERFIL = "INSERT INTO permiso_perfil(" +
        "per_codigo_perfil, " +
        "pes_codigo_permiso) " +
        "VALUES (?, ?) ";
      private String TX_DELETE_PERMISO_PERFIL = "DELETE FROM permiso_perfil " +
        "WHERE per_codigo_perfil = ? " +
        "AND pes_codigo_permiso = ? ";
      public ResultSet ListarPermisosPerfil(CMPPermiso permisoBuscado, Perfil
perfilBuscado) throws SQLException
      {
            String sql = SQL_SELECT_PERMISO_PERFIL;
```

```
            boolean set = false;
            Collection<DalParameter> parms = new ArrayList<DalParameter>();
            int index = 1;
            if (perfilBuscado.getCodigoPerfil() > 0)
            {
                  sql += "WHERE per_codigo_perfil LIKE CONCAT('%', ?, '%') ";
                  DalParameter parm = new DalParameter();
                  parm.setIndex(index);
                  index++;
                  parm.setType(ParameterType.INTEGER);
      parm.setValue(String.valueOf(perfilBuscado.getCodigoPerfil()));
                  parms.add(parm);
                  set = true;
            }
            if (permisoBuscado.getCodigoPermiso() > 0)
            {
                  if (set)
                        sql += "AND pes_codigo_permiso LIKE CONCAT('%', ?, '%')
";
                  else
                        sql += "WHERE pes_codigo_permiso LIKE CONCAT('%', ?,
'%') ";
                  DalParameter parm = new DalParameter();
                  parm.setIndex(index);
                  parm.setType(ParameterType.INTEGER);
            parm.setValue(String.valueOf(permisoBuscado.getCodigoPermiso()));
                  parms.add(parm);
            }
            PreparedStatement sta = Herramientas.getConn().prepareStatement(sql);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            ResultSet rs = sta.executeQuery();
            return rs;
      }
      public void InsertarPermisoPerfil(CMPPermiso permisoAInsertar, Perfil
perfilAInsertar) throws SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter codigoPerfil = new DalParameter();
            codigoPerfil.setIndex(1);
            codigoPerfil.setType(ParameterType.INTEGER);
```

```
        codigoPerfil.setValue(String.valueOf(perfilAInsertar.getCodigoPerfil()));
                parms.add(codigoPerfil);
                DalParameter codigoPermiso = new DalParameter();
                codigoPermiso.setIndex(2);
                codigoPermiso.setType(ParameterType.INTEGER);
        codigoPermiso.setValue(String.valueOf(permisoAInsertar.getCodigoPermiso()));
                parms.add(codigoPermiso);
                PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_INSERT_PERMISO_PERFIL);
                sta = Herramientas.getPreparedStatementReady(sta, parms);
                sta.executeUpdate();
        }
        public void EliminarPermisoPerfil(CMPPermiso permisoAEliminar, Perfil
perfilAEliminar) throws SQLException
        {
                Collection<DalParameter> parms  = new ArrayList<DalParameter>();
                DalParameter codigoPerfil = new DalParameter();
                codigoPerfil.setIndex(1);
                codigoPerfil.setType(ParameterType.INTEGER);
        codigoPerfil.setValue(String.valueOf(perfilAEliminar.getCodigoPerfil()));
                parms.add(codigoPerfil);
                DalParameter codigoPermiso = new DalParameter();
                codigoPermiso.setIndex(2);
                codigoPermiso.setType(ParameterType.INTEGER);
        codigoPermiso.setValue(String.valueOf(permisoAEliminar.getCodigoPermiso()));
                parms.add(codigoPermiso);
                PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_DELETE_PERMISO_PERFIL);
                sta = Herramientas.getPreparedStatementReady(sta, parms);
                sta.executeUpdate();
        }
}
```

Ilustración 4.4: Código de la clase PermisoPerfilBdd, capa de acceso a datos para el objeto "PermisoPerfil"

```
package biblioteca.bdd;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
```

```
import java.util.Collection;
import biblioteca.dal.DalParameter;
import biblioteca.dal.ParameterType;
import biblioteca.model.Usuario;;
public class UsuarioBdd {
	private String SQL_SELECT_USUARIO = "SELECT codigo_usuario,
per_codigo_perfil, " +
		"nombre_usuario, apellido_usuario, " +
		"telefono_usuario, nombre_acceso, " +
		"clave_acceso, estado_usuario " +
		"FROM usuario ";
	private String TX_INSERT_USUARIO = "INSERT INTO usuario(" +
		"codigo_usuario, per_codigo_perfil, " +
		"nombre_usuario, apellido_usuario, " +
		"telefono_usuario, nombre_acceso, " +
		"clave_acceso, estado_usuario) " +
		"VALUES (?, ?, ?, ?, ?, ?, ?, ?) ";
	private String TX_UPDATE_USUARIO = "UPDATE usuario " +
		"SET per_codigo_perfil =?, " +
		"nombre_usuario = ?, " +
		"apellido_usuario = ?, " +
		"telefono_usuario =?, " +
		"nombre_acceso = ?, " +
		"clave_acceso = ?, " +
		"estado_usuario = ? " +
		"WHERE codigo_usuario = ?";
	private String TX_DELETE_USUARIO = "DELETE FROM usuario " +
		   "WHERE codigo_usuario = ?";
	public ResultSet ListarUsuarios(Usuario usuarioBuscado) throws SQLException
	{
		String sql = SQL_SELECT_USUARIO;
		boolean set = false;
		Collection<DalParameter> parms = new ArrayList<DalParameter>();
		int index = 1;
	if (usuarioBuscado.getCodigoUsuario() > 0)
		{
			sql += "WHERE codigo_usuario LIKE CONCAT('%', ?, '%') ";
			DalParameter parm = new DalParameter();
			parm.setIndex(index);
			index++;
```

```
                    parm.setType(ParameterType.INTEGER);
        parm.setValue(String.valueOf(usuarioBuscado.getCodigoUsuario()));
                    parms.add(parm);
                    set = true;
                }
                if (usuarioBuscado.getCodigoPerfil() > 0)
                {
                    if (set)
                        sql += "AND per_codigo_perfil LIKE CONCAT('%', ?, '%') ";
                    else
                    {
                        sql += "WHERE per_codigo_perfil LIKE CONCAT('%', ?, '%')
";
                        set = true;
                    }
                    DalParameter parm = new DalParameter();
                    parm.setIndex(index);
                    index++;
                    parm.setType(ParameterType.INTEGER);
                    parm.setValue(String.valueOf(usuarioBuscado.getCodigoPerfil()));
                    parms.add(parm);
                    set = true;
                }
                if (!usuarioBuscado.getNombreUsuario().isEmpty())
                {
                    if (set)
                        sql += "AND nombre_usuario LIKE CONCAT('%', ?, '%') ";
                    else
                    {
                        sql += "WHERE nombre_usuario LIKE CONCAT('%', ?, '%') ";
                        set = true;
                    }
                    DalParameter parm = new DalParameter();
                    parm.setIndex(index);
                    index++;
                    parm.setType(ParameterType.STRING);
                    parm.setValue(usuarioBuscado.getNombreUsuario());
                    parms.add(parm);
                }
```

```
            if (!usuarioBuscado.getApellidoUsuario().isEmpty())
            {
                  if (set)
                        sql += "AND apellido_usuario LIKE CONCAT('%', ?, '%') ";
                  else
                  {
                        sql += "WHERE apellido_usuario LIKE CONCAT('%', ?, '%')
";
                        set = true;
                  }
                  DalParameter parm = new DalParameter();
                  parm.setIndex(index);
                  index++;
                  parm.setType(ParameterType.STRING);
                  parm.setValue(usuarioBuscado.getApellidoUsuario());
                  parms.add(parm);
            }
            if (!usuarioBuscado.getTelefonoUsuario().isEmpty())
            {
                  if (set)
                        sql += "AND telefono_usuario LIKE CONCAT('%', ?, '%') ";
                  else
                  {
                        sql += "WHERE telefono_usuario LIKE CONCAT('%', ?, '%')
";
                        set = true;
                  }
                  DalParameter parm = new DalParameter();
                  parm.setIndex(index);
                  index++;
                  parm.setType(ParameterType.STRING);
                  parm.setValue(usuarioBuscado.getTelefonoUsuario());
                  parms.add(parm);
            }
            if (!usuarioBuscado.getNombreAcceso().isEmpty())
            {
                  if (set)
                        sql += "AND nombre_acceso LIKE CONCAT('%', ?, '%') ";
                  else
                  {
```

```
                        sql += "WHERE nombre_acceso LIKE CONCAT('%', ?, '%') ";
                        set = true;
                }
                DalParameter parm = new DalParameter();
                parm.setIndex(index);
                index++;
                parm.setType(ParameterType.STRING);
                parm.setValue(usuarioBuscado.getNombreAcceso());
                parms.add(parm);
        }
        if (set)
                sql += "AND estado_usuario LIKE CONCAT('%', ?, '%') ";
        else
                sql += "WHERE estado_usuario LIKE CONCAT('%', ?, '%') ";
        DalParameter parm = new DalParameter();
        parm.setIndex(index);
        parm.setType(ParameterType.BOOLEAN);
        parm.setValue(String.valueOf(usuarioBuscado.isEstadoCuenta()));
        parms.add(parm);
        PreparedStatement sta = Herramientas.getConn().prepareStatement(sql);
        sta = Herramientas.getPreparedStatementReady(sta, parms);
        ResultSet rs = sta.executeQuery();
        return rs;
}
public void InsertarUsuario(Usuario usuarioAInsertar) throws SQLException
{
        Collection<DalParameter> parms  = new ArrayList<DalParameter>();
        DalParameter codigoUsuario = new DalParameter();
        codigoUsuario.setIndex(1);
        codigoUsuario.setType(ParameterType.INTEGER);
codigoUsuario.setValue(String.valueOf(usuarioAInsertar.getCodigoUsuario()));
        parms.add(codigoUsuario);
        DalParameter codigoPerfil = new DalParameter();
        codigoPerfil.setIndex(2);
        codigoPerfil.setType(ParameterType.INTEGER);
codigoPerfil.setValue(String.valueOf(usuarioAInsertar.getCodigoPerfil()));
        parms.add(codigoPerfil);
        DalParameter nombreUsuario = new DalParameter();
        nombreUsuario.setIndex(3);
        nombreUsuario.setType(ParameterType.STRING);
```

```
            nombreUsuario.setValue(usuarioAInsertar.getNombreUsuario());
            parms.add(nombreUsuario);
            DalParameter apellidoUsuario = new DalParameter();
            apellidoUsuario.setIndex(4);
            apellidoUsuario.setType(ParameterType.STRING);
            apellidoUsuario.setValue(usuarioAInsertar.getApellidoUsuario());
            parms.add(apellidoUsuario);
            DalParameter telefonoUsuario = new DalParameter();
            telefonoUsuario.setIndex(5);
            telefonoUsuario.setType(ParameterType.STRING);
            telefonoUsuario.setValue(usuarioAInsertar.getTelefonoUsuario());
            parms.add(telefonoUsuario);
            DalParameter nombreAcceso = new DalParameter();
            nombreAcceso.setIndex(6);
            nombreAcceso.setType(ParameterType.STRING);
            nombreAcceso.setValue(usuarioAInsertar.getNombreAcceso());
            parms.add(nombreAcceso);
            DalParameter claveAcceso = new DalParameter();
            claveAcceso.setIndex(7);
            claveAcceso.setType(ParameterType.STRING);
            claveAcceso.setValue(usuarioAInsertar.getClaveAcceso());
            parms.add(claveAcceso);
            DalParameter estadoUsuario = new DalParameter();
            estadoUsuario.setIndex(8);
            estadoUsuario.setType(ParameterType.BOOLEAN);
      estadoUsuario.setValue(String.valueOf(usuarioAInsertar.isEstadoCuenta()));
            parms.add(estadoUsuario);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_INSERT_USUARIO);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
      }
      public void ActualizarUsuario(Usuario usuarioAActualizar) throws SQLException
      {
            Collection<DalParameter> parms  = new ArrayList<DalParameter>();
            DalParameter codigoPerfil = new DalParameter();
            codigoPerfil.setIndex(1);
            codigoPerfil.setType(ParameterType.INTEGER);
      codigoPerfil.setValue(String.valueOf(usuarioAActualizar.getCodigoPerfil()));
            parms.add(codigoPerfil);
```

```
            DalParameter nombreUsuario = new DalParameter();
            nombreUsuario.setIndex(2);
            nombreUsuario.setType(ParameterType.STRING);
            nombreUsuario.setValue(usuarioAActualizar.getNombreUsuario());
            parms.add(nombreUsuario);
            DalParameter apellidoUsuario = new DalParameter();
            apellidoUsuario.setIndex(3);
            apellidoUsuario.setType(ParameterType.STRING);
            apellidoUsuario.setValue(usuarioAActualizar.getApellidoUsuario());
            parms.add(apellidoUsuario);
            DalParameter telefonoUsuario = new DalParameter();
            telefonoUsuario.setIndex(4);
            telefonoUsuario.setType(ParameterType.STRING);
            telefonoUsuario.setValue(usuarioAActualizar.getTelefonoUsuario());
            parms.add(telefonoUsuario);
            DalParameter nombreAcceso = new DalParameter();
            nombreAcceso.setIndex(5);
            nombreAcceso.setType(ParameterType.STRING);
            nombreAcceso.setValue(usuarioAActualizar.getNombreAcceso());
            parms.add(nombreAcceso);
            DalParameter claveAcceso = new DalParameter();
            claveAcceso.setIndex(6);
            claveAcceso.setType(ParameterType.STRING);
            claveAcceso.setValue(usuarioAActualizar.getClaveAcceso());
            parms.add(claveAcceso);
            DalParameter estadoUsuario = new DalParameter();
            estadoUsuario.setIndex(7);
            estadoUsuario.setType(ParameterType.BOOLEAN);
      estadoUsuario.setValue(String.valueOf(usuarioAActualizar.isEstadoCuenta()));
            parms.add(estadoUsuario);
            DalParameter codigoUsuario = new DalParameter();
            codigoUsuario.setIndex(8);
            codigoUsuario.setType(ParameterType.INTEGER);
      codigoUsuario.setValue(String.valueOf(usuarioAActualizar.getCodigoUsuario()))
;
            parms.add(codigoUsuario);
            PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_UPDATE_USUARIO);
            sta = Herramientas.getPreparedStatementReady(sta, parms);
            sta.executeUpdate();
```

```
        }
        public void EliminarUsuario(Usuario usuarioAEliminar) throws SQLException
        {
                Collection<DalParameter> parms  = new ArrayList<DalParameter>();
                DalParameter codigoUsuario = new DalParameter();
                codigoUsuario.setIndex(1);
                codigoUsuario.setType(ParameterType.INTEGER);
        codigoUsuario.setValue(String.valueOf(usuarioAEliminar.getCodigoUsuario()));
                parms.add(codigoUsuario);
                PreparedStatement sta =
Herramientas.getConn().prepareStatement(TX_DELETE_USUARIO);
                sta = Herramientas.getPreparedStatementReady(sta, parms);
                sta.executeUpdate();
        }
}
```

Ilustración 4.5: Código de la clase UsuarioBdd, capa de acceso a datos para el objeto "Usuario"

Capa de herramientas**:** En las Ilustraciones 4.6 y 4.7 se presenta el código fuente de los patrones de diseño Command, en los que se pueden observar las funciones de gestión del objeto "Permisos" y "Perfiles", estas funciones están separadas del objeto gestionado, y a su vez consumen las funciones de la capa de acceso a datos. El patrón command se ha definido como interfaz, de manera que se lo pueda implementar de acuerdo a las prestaciones de la capa de datos, en este caso se utilizará MySQL.

```
package biblioteca.services;
import java.util.Collection;
import biblioteca.model.CMPPermiso;
import biblioteca.model.Perfil;
public interface CMMPerfil {
        public Collection<Perfil> BuscarPerfiles(Perfil perfilABuscar);
        public Perfil CrearPerfil(int codigoPerfil, String descripcionPerfil,
                        Collection<CMPPermiso> permisos);
        public Collection<CMPPermiso> BuscarPermisos(Perfil perfil);
        public boolean RegistrarPerfil(Perfil perfilARegistrar);
        public boolean AsignarPermisos(Perfil perfil, Collection<CMPPermiso>
permisos);
        public boolean EliminarPermisos(Perfil perfil, Collection<CMPPermiso>
```

```
permisos);
       public boolean ModificarPerfiles(Collection<Perfil> perfilesAModificar);
       public boolean EliminarPerfiles(Collection<Perfil> perfilesAEliminar);
}
```

Ilustración 4.6: Código de la Patrón CMMPerfil, de tipo Command

```
package biblioteca.services;
import java.util.Collection;
import biblioteca.model.CMPPermiso;
import biblioteca.model.PermisoCompuesto;
public interface CMMPermisos {
       public Collection<CMPPermiso> BuscarPermisosRaiz(CMPPermiso
permisoABuscar);
       public PermisoCompuesto BuscarPermisoPadre(CMPPermiso permisoHijo);
       public Collection<CMPPermiso> BuscarPermisosHijo(PermisoCompuesto
permisoPadre);
       public CMPPermiso CrearPermiso();
}
```

Ilustración 4.7: Código de la Patrón CMMPermiso, de tipo Command

Capa de Abstracción: En la Ilustración 4.3 se presenta el patrón de diseño Composite, en el se pueden observar los atributos que definirán al objeto Permiso, independientemente de si se trata de un objeto simple o de uno compuesto.

```
public abstract class CMPPermiso {
       public int getCodigoPermiso() {
              return 0;
       }
       public String getDescripcionPermiso() {
              return null;
       }
}
```

Ilustración 4.8: Definición de la clase abstracta "Permiso" (Patrón de tipo Composite)

Objeto Simple: Pertenece a la capa de abstracción, hereda de la clase compuesta CMPPermiso, de esta manera, los objetos que gestionan los permisos pueden manipular objetos compuestos u objetos simples, utilizando los mismo algoritmos.

```
public class PermisoSimple extends CMPPermiso
{
	private int codigoPermiso;
	private String descripcionPermiso;
	private PermisoCompuesto padre;
	public PermisoCompuesto getPadre() {
		return padre;
	}
	public void setPadre(PermisoCompuesto padre) {
		this.padre = padre;
	}
	public int getCodigoPermiso() {
		return codigoPermiso;
	}
	public void setCodigoPermiso(int codigoPermiso) {
		this.codigoPermiso = codigoPermiso;
	}
	public String getDescripcionPermiso() {
		return descripcionPermiso;
	}
	public void setDescripcionPermiso(String valorPermiso) {
		this.descripcionPermiso = valorPermiso;
	}
}
```

Ilustración 4.9: Implementación del Patrón composite, para un objeto simple

***Objeto Compuesto*:** Pertenece a la capa de abstracción, hereda de la clase compuesta CMPPermiso, en este objeto podemos notar que existe una lista de objetos CMPPermiso, de manera que puede albergar, como hijos, a objetos compuestos o simples.

```
public class PermisoCompuesto extends CMPPermiso
{
	private int codigoPermiso;
	private String descripcionPermiso;
	private Collection<CMPPermiso> permisos;
	public Collection<CMPPermiso> getPermisos()
	{
```

```
            return permisos;
    }
    public void setPermisos(Collection<CMPPermiso> permisos)
    {
            this.permisos = permisos;
    }
    public void agregarPermiso (CMPPermiso permiso)
    {


    }
    public CMPPermiso eliminarPermiso (int codigoPermiso)
    {
            return null;
    }
    public CMPPermiso buscarPermiso (int codigoPermiso)
    {
            return null;
    }
    public int getCodigoPermiso() {
            return codigoPermiso;
    }
    public void setCodigoPermiso(int codigoPermiso) {
            this.codigoPermiso = codigoPermiso;
    }
    public String getDescripcionPermiso() {
            return descripcionPermiso;
    }
    public void setDescripcionPermiso(String descripcionPermiso) {
            this.descripcionPermiso = descripcionPermiso;
    }
}
```

Ilustración 4.10: Implementación del Patrón composite, para un objeto compuesto

4.3. ANÁLISIS DEL USO DE LA GUÍA

El uso de la guía permitió la exitosa inclusión de 4 patrones de diseño a la aplicación seleccionada para el caso de estudio, con lo cual se mejoraron principalmente los aspectos relacionados con Escalabilidad, Seguridad, Modularidad, y Calidad del

sistema; además permitió optimizar los tiempos de desarrollo de la misma al reciclar la lógica de programación de los patrones de diseño implementados.

La escalabilidad de la aplicación se mejora gracias a que se utilizan clases abstractas e interfaces a la hora de relacionar objetos, de esta manera se puede agregar nuevos objetos, que implementen dichas interfaces, sin afectar a las clases que manipulan esos objetos.

La seguridad de la aplicación se mejoró gracias a la inclusión del patrón Singleton, en este caso el patrón gestiona completamente el acceso a la base de datos, evitando que existan conexiones paralelas, el uso de una capa de acceso a datos permite controlar la parametrización de consultas y transacciones SQL, evitando la inyección de código SQL.

El uso de patrones de diseño mejora la modularidad del sistema, permite mejorar la cohesión entre objetos, reduciendo notablemente su acoplamiento, haciendo factible la modificación de funcionalidad existente, la inclusión de nueva funcionalidad y el reemplazo de la misma por una nueva. Es importante indicar que el caso de estudio presento las prestaciones necesarias para poder aplicar la guía desarrollada, sin embargo, ésta guía presenta además opciones útiles a la hora de estudiar los patrones de diseño, individual y colectivamente.

CAPÍTULO V

5. CONCLUSIONES Y RECOMENDACIONES

5.1. CONCLUSIONES

- Los patrones de diseño constituyen una valiosa herramienta para la comprensión y entendimiento de la programación orientada a objetos, de ahí que tras veinte años de su invención, siguen siendo utilizados en el desarrollo de software y, más aún, marcando pautas para nuevas tendencias en programación.

- Los patrones de diseño facilitan el mantenimiento de software y correctamente aplicados desde el diseño de un sistema, pueden mejorar altamente las prestaciones de escalabilidad, mantenimiento y seguridad del software.

- Los patrones de diseño pueden optimizar los tiempos de desarrollo de una aplicación nueva, pues forman parte de librerías reutilizables.

- En este trabajo el entendimiento de patrones de diseño se facilita mediante ejemplos prácticos orientados a la ingeniería de software, pues generalmente en la bibliografía se encuentran ejemplos que hacen uso de analogías, lo cual puede complicar el estudio de los patrones.

- El uso descuidado de los patrones de diseño puede desatar una serie de problemas que pueden provocar retrasos en el desarrollo de los proyectos, dificultad en la reutilización de los módulos y problemas en la mantenibilidad; por lo cual no se debe incluir patrones de manera indiscriminada en un sistema de Software.

5.2. RECOMENDACIONES

- Se recomienda la inclusión de patrones de diseño en el desarrollo de software por necesidad y para solucionar problemas presentes en el sistema en desarrollo; pero no agregarlos sin mayor conocimiento de los mismos.

- Es recomendable que cuando se desee incluir un patrón de diseño en un sistema, se debe, en lo posible, consultar con alguna persona con experiencia en el manejo de patrones de diseño, pues su uso apropiado demanda tener cierta experiencia al respecto.

- También es recomendable que en el desarrollo de Software, se construya modelos lo más simples; además emplear los patrones más adecuados y fácilmente entendibles, pues son aspectos claves para desarrollar aplicaciones de calidad.

BIBLIOGRAFÍA

TESIS:

ROBALINO, Juan; RUILOVA, Carolina. Uso de los patrones de diseño en el desarrollo de Aplicaciones de Software. 2006.

PROAÑO, Edith; SARANGO, María. Guía para el desarrollo de Aplicaciones Robustas Utilizando Técnicas y Estrategias de Seguridad. 2005.

GÓMEZ, Juan; GUERRA, Fernando. Reingeniería del sistema de calificación automática de exámenes. 2008

LIBROS:

BISHOP, Judith. C# 3.0 Design Patterns. O'Reilly. California – Estados Unidos. Diciembre 2007.

COOPER, James W. The Design Patterns Java Companion. Addison-Wesley. Octubre 1998.

HOLZNER, Steve. Design Patterns for dummies. Wiley Publishing, Inc. Canada. 2006.

DIAZ, Dustin; HARMES, Ross. Pro JavaScript Design Patterns. Apress, Estados Unidos, 2008.

ARTÍCULOS:

IEEE. Using Design Patterns & Frameworks to Develop Object- Oriented Communication Systems. 1998.

IEEE, Synthesizing evocative imagery through design patterns. 2002

PÁGINAS WEB:

ANGEL, Ramiro. http://www.proactiva-calidad.com/java/patrones/index.html#algunos_patrones. Mayo 2008

CIBERAULA. http://java.ciberaula.com/articulo/diseno_patrones_j2ee/

SCRIBD. http://www.scribd.com/doc/3930805/Patrones-de-Diseno

SCRIBD. http://www.scribd.com/doc/6819691/Rose-creating-Usecase-and-Classdiagram

OSMOSIS LATINA. http://javaejb.osmosislatina.com/curso/patrones.htm

EMAGISTER. http://mit.ocw.universia.net/6.170/6.170/f01/pdf/lecture-12.pdf

GALINUS. http://www.galinus.com/es/articulos/ejemplo-patrones-diseno-interaccion.html

DOMINGUEZ, Jorge. http://www.stackframe.net/es/content/01-2009/patrones-de-diseno-introduccion

GRACIAS, Joaquín. http://www.ingenierosoftware.com/analisisydiseno/patrones-diseno.php

PROGRAMACIÓN. http://www.chuidiang.com/ood/index.php

LAGOS, Manuel. http://www.elrincondelprogramador.com/default.asp?id=29&pag= articulos/leer.asp

RINCÓN DEL PROGRAMADOR. http://www.info-ab.uclm.es/asignaturas/42579/pdf/04-Capitulo4a.pdf

RINCÓN DEL PROGRAMADOR. http://www.elrincondelprogramador.com/default.asp?id= 45&pag=articulos%2Fleer.asp

LAGO, Ramiro. http://www.proactiva-calidad.com/java/patrones/index.html

TEDESCHI , Nicolás. http://msdn.microsoft.com/es-es/library/bb972240.aspx

CARUSO, Javier. http://apyd.blogspot.com/

WIKIPEDIA. http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o

WIKIPEDIA. http://es.wikipedia.org/wiki/Abstract_Factory_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Builder_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Factory_Method_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Prototype_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o_Singleton

WIKIPEDIA. http://es.wikipedia.org/wiki/Adapter_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Bridge_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Composite_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Decorator_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Facade_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Flyweight_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Proxy_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA.http://es.wikipedia.org/wiki/Chain_of_Responsibility_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Command_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Interpreter_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Iterator_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Mediator_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Memento_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/State_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Strategy_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA.http://es.wikipedia.org/wiki/Template_Method_(patr%C3%B3n_de_dise%C3%B1o)

WIKIPEDIA. http://es.wikipedia.org/wiki/Visitor_(patr%C3%B3n_de_dise%C3%B1o)

WORDREFERENCE. http://www.wordreference.com/definicion/acoplamiento

WORDREFERENCE. http://www.wordreference.com/definicion/cohesión

WORDREFERENCE . http://www.wordreference.com/definicion/escalabilidad

WORDREFERENCE. http://www.wordreference.com/definicion/mantenimiento

WORDREFERENCE. http://www.wordreference.com/definicion/reutilización

WORDREFERENCE. http://www.wordreference.com/definicion/dinamismo

WORDREFERENCE. http://www.wordreference.com/definicion/redundancia

WORDREFERENCE. http://es.wikipedia.org/wiki/Granularidad

GLOSARIO

Patrón: Un patrón es considerado un método estructurado o una plantilla estándar para describir una serie de buenas prácticas.

Patrón de Diseño: Un patrón de diseño es una solución a un problema de diseño, que posee ciertas características estándares, dentro de las cuales se puede resaltar su efectividad al resolver problemas similares en ocasiones anteriores.

Enterprise Library: Enterprise Library, una herramienta de desarrollo gratuita que facilita la incorporación de las empresas a la plataforma .NET. Gracias a sus siete bloques de aplicaciones (Caching, Configuration, Cryptography, Data Access, Exception Handling, Logging y Security), simplifica los procesos de acceso a datos, administración y manejo de excepciones, configuración y encriptación para todo tipo de desarrollos.

JVM: JVM es un programa nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el Java bytecode), el cual es generado por el compilador del lenguaje Java.

ANEXO 1: CASO DE ESTUDIO

CD adjunto: contiene la aplicación