Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

REUTILIZACIÓN DEL SOFTWARE, METODOLOGÍAS Y APLICACIONES MÁS FRECUENTES

Carlos Estuardo Figueroa Santiago
Asesorado por el Ing. Calixto Raul Monzon Pérez

Guatemala, mayo de 2010

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

REUTILIZACIÓN DEL SOFTWARE, METODOLOGÍAS Y APLICACIONES MÁS FRECUENTES

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA
DE LA FACULTAD DE INGENIERÍA
POR:

CARLOS ESTUARDO FIGUEROA SANTIAGO
ASESORADO POR EL ING. CALIXTO RAUL MONZON PÉREZ

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, MAYO DE 2010

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Inga. Glenda Patricia García Soria
VOCAL II	Inga. Alba Maritza Guerrero de López
VOCAL III	Ing. Miguel Angel Dávila Calderón
VOCAL IV	Br. Luis Pedro Ortíz de León
VOCAL V	Agr. José Alfredo Ortíz Herincx
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

TRIBUNAL QUE PRACTICÓ EL EXAMEN PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADOR	Ing. Juan Alvaro Díaz Ardavín
EXAMINADOR	Ing. Pedro Pablo Hernandez Ramírez
EXAMINADOR	Ing. Ludwing Federico Altan Sac
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos establece la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

Reutilización del software, metodologías y aplicaciones más frecuentes,

tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha 15 de julio de 2009.

Carlos Estuardo Figueroa Santiago

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas

Guatemala 23 de febrero de 2010

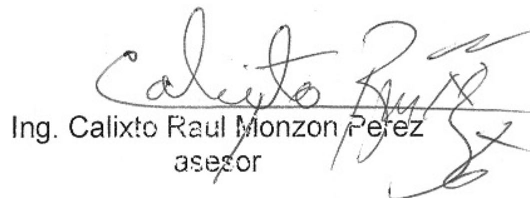
Ingeniero
Carlos Alfredo Azurdia Morales
Coordinador del Área de Trabajos de Graduación

Respetable Ingeniero Azurdia:

Por este medio le informo que luego de haber asesorado el trabajo de graduación titulado "REUTILIZACIÓN DEL SOFTWARE, METODOLOGÍAS Y APLICACIONES MÁS FRECUENTES" desarrollado por el estudiante universitario de la carrera de Ingeniería en Ciencias y Sistemas, Carlos Estuardo Figueroa Santiago, carnet 9515831, se hicieron las revisiones respectivas y a mi criterio cumple con los objetivos propuestos para su desarrollo.

Agradeciendo su atención a la presente

Atentamente



Ing. Calixto Raul Monzon Perez
asesor

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 10 de Marzo de 2010

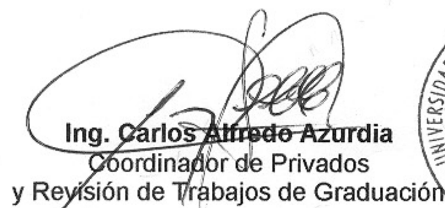
Ingeniero
Marlon Antonio Pérez Turk
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Pérez:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **CARLOS ESTUARDO FIGUEROA SANTIAGO,** titulado: **"REUTILIZACION DEL SOFTWARE, METODOLOGIAS Y APLICACIONES MAS FRECUENTES",** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,



Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

E S C U E L A D E C I E N C I A S Y S I S T E M A S

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÌA
ESCUELA DE CIENCIAS Y SISTEMAS
TEL: 24767644

*El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, de trabajo de graduación titulado **"REUTILIZACIÓN DEL SOFTWARE, METODOLOGÍAS Y APLICACIONES MAS FRECUENTES"**, presentado por el estudiante CARLOS ESTUARDO FIGUEROA SANTIAGO, aprueba el presente trabajo y solicita la autorización del mismo.*

"ID Y ENSEÑAD A TODOS"

Ing. Marlon Antonio Pèrez Turk
Director, Escuela de Ingenierìa Ciencias y Sistemas

Guatemala, 13 de mayo 2010

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

Ref. DTG.152.2010

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al trabajo de graduación titulado; **REUTILIZACIÓN DEL SOFTWARE, METODOLOGÍAS Y APLICACIONES MÁS FRECUENTES,** presentado por el estudiante universitario **Carlos Estuardo Figueroa Santiago,** autoriza la impresión del mismo.

IMPRÍMASE.

Ing. Murphy Olympo Paiz Recinos
DECANO

Guatemala, mayo de 2010

/cc
c.c. archivo.

DEDICATORIA A:

DIOS Y A LA VIRGEN MARÍA

Por haberme dado una vida llena de bendiciones y por haberme permitido llegar a este momento de mi vida.

MIS PADRES

Dr. Carlos Alfredo Figueroa Ramirez y de forma especial a mi mamá Licda. María Asenneth Santiago de Figueroa.

MIS HERMANOS

Francisco Alfredo y María Asenet de los Angeles, por su apoyo y cariño.

MIS SOBRINOS Y CUÑADOS

Nahbi Magdiel, Iram Didier y Silvia, por su cariño y dulzura.

MIS ABUELAS

Angelina Ramirez de Figueroa (q.e.p.d) y Delia Paniagua de Gudiel, por sus oraciones en el cielo y en la tierra.

MI FAMILIA

Abuelos, tíos, tías, primos y demás familia por su apoyo y cariño.

MIS AMIGOS

Por su amistad, compañerismo y tantos momentos compartidos.

AGRADECIMIENTOS A:

DIOS Y A LA VIRGEN MARÍA

Por darme la vida, por saberme guiar por el buen camino, por no abandonarme, por permitirme llegar a este día y alcanzar este objetivo en mi vida.

MIS PADRES

Dr. Carlos Alfredo Figueroa Ramirez con su ejemplo, alegría y sabiduría; y muy especialmente a mi mamá Licda. María Asenneth Santiago de Figueroa, por formar valores y principios y mostrarme con su entereza, dedicación, esfuerzo y amor que son unas personas maravillosas, una bendición de Dios para mí.

MI ASESOR

Ing. Calixto Monzon, por su apoyo, tiempo, experiencia profesional y paciencia en revisar este trabajo de graduación.

LA UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

Por albergarme en su seno como estudiante.

LA FACULTAD DE INGENIERÍA

Por brindarme los conocimientos necesarios para graduarme como ingeniero.

PUEBLO DE GUATEMALA

Al cual los estudiantes y profesionales de la Universidad de San Carlos nos debemos.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES…………………………………………………...................... VII

GLOSARIO.. IX

RESUMEN... XVII

OBJETIVOS XIX

INTRODUCCIÓN... XXI

1 MARCO TEÓRICO: CONCEPTOS BÁSICOS DE LA REUTILIZACIÓN….. 1

1.1 Beneficios de la reutilización ………………………………………………. 1

1.1.1 Tiempo ……………………………………………………………………. 1

1.1.2 Calidad …………………………………………………………….....…. 1

1.1.3 Estandarización ………………………………………………………. 2

1.2 Barreras a la reutilización ………………………………………………..… 2

1.2.1 Elaboración, administración y uso de componentes reutilizables . 3

1.2.2 Gastos al comenzar …………………………………………….…...... 3

1.2.3 Reto a los métodos tradicionales de ingeniería de software …….. 4

1.3 Reutilización práctica de software ………………………………………… 4

1.3.1 Prácticas de desarrollo con reutilización …………………………... 5

1.3.1.1 Tecnologías orientadas a objetos …………………………..... 5

1.3.1.2 Desarrollando con reutilización reutilización oportunista….. 6

1.3.1.3 Desarrollando con reutilización: reutilización sistemática….. 7

1.3.2 Prácticas sin desarrollo con reutilización del producto ………..….. 8

2 INGENIERÍA DE DOMINIOS…………………...………………………...…. 9

2.1 Dominio ……………………………………………………………………… 9

2.1.1 Definición de dominio ………………………………………………... 9

2.1.2 Relación entre dominios y aplicaciones …………………………... 11

2.1.3 Dominios verticales y horizontales ………………………………… 11

2.2 Ingeniería de dominios e ingeniería de aplicación……………………. 12

2.3 Análisis del dominio, diseño del dominio e implementación del dominio …………………….……………………………….……………….. 14
2.3.1 Análisis del dominio …...……………………………….…………... 14
2.3.2 Diseño del dominio ……………………………………………….…… 19
2.3.3 Implementación del dominio …….…………………………………. 20
2.4 Distinción de términos relacionados …………………………………… 21
2.4.1 Línea de producto ……………………………………………..…… 21
2.4.2 Ingeniería de requerimientos ……………………………………..… 21
2.4.3 Ingeniería de reutilización…………………………….…………….. 22
2.4.4 Arquitecturas de software específicas de dominio ………………. 23
2.4.5 Arquitectura de referencia ……………………...……………..... 23
2.5 Métodos de ingeniería de dominios …………..……………………..... 24
2.6 Ingeniería de dominios y metodologías orientadas al análisis y diseño orientados a objetos (OOA/D) …...…………………………………….... 27
2.6.1 Defectos en los métodos OOA/D existentes …………………….. 27
2.6.2 Metodologías OOA/D apoyando a la ingeniería de dominios ….. 28
2.7 Problemas emergentes ………………………………………………….. 32
3 PRINCIPIOS DEL ANÁLISIS Y DISEÑO ORIENTADO AL OBJETO PARA REUTILIZACIÓN..…………………...…………………………….…………… 35
3.1 El principio de abierto/cerrado ………………………….…………..... 35
3.1.1 Polimorfismo dinámico……………………..……. ……………….. 36
3.1.2 Polimorfismo estático …………………………………………….. 39
3.1.3 Metas arquitectónicas del PAC ………..………………………….. 40
3.2 El principio de sustitución de Liskov ………..……………………….… 40
3.2.1 El dilema del círculo / elipse ……………..………………………... 41
3.2.2 Diseño por contrato ………………...……………………………… 45
3.3 El principio de inversión de dependencia ……...…………………..….. 45
3.3.1 Mitigando fuerzas ………………………………..………….…….. 47
3.3.2 Creación del objeto …………………………….………………….... 48

3.4 El principio de separación de la interfaz ………………….………..… 48
3.4.1 ¿Qué significa específico del cliente? …………………………..... 50
3.4.2 Cambiando interfaces …………………..……………………….... 50
3.5 El principio de equivalencia reutilización/publicación…...……..…...… 51
3.6 El principio de cierre común …………………….……………………… 52
3.7 El principio de reutilización común …………………………………..... 52
3.7.1 Tensión entre los principios de cohesión de paquetes ……...…… 53
3.8 El principio de dependencia acíclica …………………………………… 54
3.8.1 Un ciclo se introduce …………………………………………….…. 55
3.8.2 Quebrando un ciclo ……..…...……………………………………... 57
3.9 El principio de las dependencias estables ……………………………... 59
3.9.1 Estabilidad …………………………………..…………………….... 60
3.9.2 Métricas de estabilidad …………………………………………….. 61
3.9.3 Fundamento …………..…………………………………………….. 62
3.10 El principio de las abstracciones estables …………………..……… 63
3.10.1 Métricas de abstracción ……………………….……..………….. 64
3.10.2 La gráfica de I versus A ……………………..……………….... 65
3.10.3 Métricas de distancia ……………………….…………………… 67
4 MÉTODOS DE REUTILIZACIÓN DE SOFTWARE …………….....…….. 69
4.1 Métodos generativos de reutilización …………………..…………...… 69
4.1.1 Sistemas basados en lenguaje ……………………..…………..… 70
4.1.2 Generadores de aplicación ………………………………………... 71
4.1.3 Sistemas transformacionales ………………………………………. 72
4.2 Métodos composicionales de reutilización ……………………………..… 73
4.2.1 Reutilización dentro de la metodología orientada a objetos ……. 82
4.2.2 Reusabilidad de objetos …………………………………………… 83
4.2.3 Reutilización de clases ……………………………………………... 83
4.2.4 Reutilización en el mundo ………………………………………….. 86
4.2.5 Marcos de trabajo de aplicación …………………………………... 87

4.2.6 Patrones de diseño …………………………………………………….… 89
5 EJEMPLOS DE REUTILIZACIÓN COMPOSICIONAL CON PATRONES DE DISEÑO EN PHP…………..…………………….....……………………….. 93
5.1 Uso del patrón de diseño observer ………..…………………………..…. 93
5.1.1 El problema …………………………..…………………………….... 93
5.1.2 La solución …………………………..…………………………….... 94
5.1.3 Código de ejemplo …………………..…………………………..…. 94
5.1.4 Explicación del ejemplo ………………..…………………………... 97
5.2 Uso del patrón de diseño factory ……………………………………..…. 99
5.2.1 El problema …………………………..…………………………….... 99
5.2.2 La solución …………………………...…………………………….... 99
5.2.3 Código de ejemplo ……………………...……………………….…... 100
5.2.4 Explicación del ejemplo ………………...………………………....... 105
5.3 Uso del patrón strategy ……………………...……………………......... 106
5.3.1 El problema ………………..…………...……………………...…. 107
5.3.2 La solución ……………………..…………..……………………..…. 107
5.3.3 Código de ejemplo ……...………………...…………………...…. 108
5.3.4 Explicación del ejemplo ……………………..………………….. 112
5.4 Uso del patrón de diseño adapter ……………………………………..…. 113
5.4.1 El problema …..……………………………………………………113
5.4.2 La solución ………………..…………………………………..…113
5.4.3 Código de ejemplo ……………………………………..…………..…114
5.4.4 Explicación del ejemplo ……………………………………..……..…119
5.5 Uso del patrón de diseño singleton ……………………………...……..... 120
5.5.1 El problema ……………………………………………..………. 121
5.5.2 La solución …………………………………………………………… 121
5.5.3 Código de ejemplo …………..………………………………………… 121
5.5.4 Segundo ejemplo ………….……………………………………... 124
5.5.5 Explicación del segundo ejemplo ……………………..………... 125

CONCLUSIONES ………………………………………………………………..… 127
RECOMENDACIONES ……………………………...……………………..... 129
BIBLIOGRAFÍA ……………………………………...………………………. 131

ÍNDICE DE ILUSTRACIONES

FIGURAS

1	Logueo cerrado para modificación	39
2	Esquema de principio de sustitución de Liskov	38
3	Dilema del círculo y el elipse	42
4	Declaración del elipse	43
5	Estructura de dependencia de una arquitectura procedural	46
6	Estructura de dependencia de una arquitectura orientada a objetos	47
7	Servicio grande con interfaces integradas	49
8	Interfaces separadas	50
9	Red de paquetes acíclicos	55
10	Un ciclo ha sido agregado	56
11	Ciclo quebrado	58
12	Otra técnica	59
13	X es un paquete estable	60
14	Y es un paquete inestable	61
15	Violación del principio de dependencias estables	63
16	Gráfica de A versus I	65
17	¿Dónde debería ir X en la gráfica A vs I?	66
18	ListaClientes y su ListaSuscripcion con bitácora agregada	96
19	Las clases Registro y FabricaRegistro	101
20	Relación entre objetos ElegidorCarro, MedidorCarro y Carro	108
21	El adaptador situado entre la gráfica y los datos	115
22	UML para el singleton manejador de base de datos	122

TABLA

I	Documento de elementos	16

GLOSARIO

ADA:[1] Ada es un lenguaje de programación estructurado y fuertemente tipado de forma estática que fue diseñado por Jean Ichbiah de CII Honeywell Bull por encargo del Departamento de Defensa de los Estados Unidos. Es un lenguaje multipropósito, orientado a objetos y concurrente, pudiendo llegar desde la facilidad de Pascal hasta la flexibilidad de C++.

API:[2] Application Programming Interface. Una interfaz de programación de aplicaciones o API (del inglés) es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. Usados generalmente en las bibliotecas.

CCM: CORBA Component Model. Modelo de componentes CORBA, es una especificación para crear del lado del servidor aplicaciones escalables, neutrales al lenguaje, transaccionales, multiusuario y seguras a nivel de empresas.

COM:[3] Component Object Model. Es una plataforma de Microsoft para componentes de software introducida por dicha empresa en 1993. Esta plataforma es utilizada para permitir la comunicación entre procesos y la creación dinámica de objetos, en cualquier lenguaje de programación que soporte dicha tecnología. El término COM

---

[1] http://es.wikipedia.org/wiki/Ada_%28lenguaje_de_programaci%C3%B3n%29
[2] http://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones
[3] http://es.wikipedia.org/wiki/Component_Object_Model

es a menudo usado en el mundo del desarrollo de software como un término que abarca las tecnologías OLE, OLE Automation, ActiveX, COM+ y DCOM. Si bien COM fue introducido en 1993, Microsoft no hizo énfasis en el nombre COM hasta 1997.

CORBA:[4] Common Object Request Broker Architecture. Arquitectura común de intermediarios en peticiones a objetos, es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. CORBA se puede considerar como un formato de documentación legible por la máquina, similar a un archivo de cabeceras, pero con más información.

DAGAR: Domain Architecture-based Generation for Ada Reuse. Generación de dominio basada en arquitectura para reutilización de ADA, es un proceso desarrollado por el equipo de Loral Defense Systems-East STARS. DAGAR es un proceso repetible y documentado acompañado por herramientas, incluye el desarrollo de una arquitectura de dominio, implementación de activos dentro de la arquitectura, y aplicación del dominio a través de la selección de activos para una aplicación particular.

FeatuRSEB: Featured Reuse-driven Software Engineering Business. Es un proceso basado en casos de uso para desarrollo de sistemas grandes de compañías, como una combinación de RSEB y FODA para identificar los problemas específicos de la familia de sistemas. FeatuRSEB definió casos de uso ampliados en la fase de ingeniería de requerimientos. Con un punto de variación en los

[4] http://es.wikipedia.org/wiki/CORBA

casos de uso, FeatuRSEB modela opciones de diferente comportamiento. Después de la fase de modelado el desarrollador tiene que seleccionar el comportamiento necesario para obtener una instancia válida del modelo.

FODA: Feature-Oriented Domain Análisis. Análisis de dominio orientado a las características, es un método de análisis de dominio que se enfoca en las “características” de los sistemas del dominio. Una característica es, de acuerdo a FODA, un aspecto, cualidad, o rasgo de un sistema que es visible al usuario final. La transmisión de una auto (automático o manual) es un ejemplo de una característica.

FORM : Feature-Oriented Reuse Method. Método de reutilización orientado a las características, es un método sistemático que busca y captura cosas en común y diferencias de aplicaciones en un dominio en términos de “características” y utilizando los resultados de los análisis para desarrollar arquitecturas de dominios y componentes. El modelo que captura las cosas en común y las diferencias es llamado “modelo de características” y es usado para apoyar tanto a la ingeniería de artefactos reutilizables de dominio como al desarrollo de aplicaciones que usan los artefactos de dominio. Una vez que un dominio es descrito y explicado en términos de “unidades” comunes y diferentes de computación, son utilizados para construir diferentes configuraciones factibles de arquitecturas reutilizables.

GLITTER: Es un sistema transformacional que codifica en reglas transformacionales decisiones que un desarrollador de software ha hecho durante sus aplicaciones.

JIAWG: The Joint Integrated Avionics Working Group. Grupo de trabajo conjunto integrado de aviación es un esfuerzo ordenado por el congreso de EEUU que ha sido formado para coordinar tecnología, compartir datos, reducir duplicación y resolver problemas comunes de los componentes.

JODA: JIAWG Oriented Domain Anaysis. Es un método que es utilizado por el JIAWG para el análisis del dominio. Este método está basado en el análisis orientado a objetos y la notación definida por Coad y Yourdon, las cuales son referidas como Coad Yourdon Object Oriented Análisis (CYOOA). Este método adiciona ciertas extensiones ya que la notación original está hecha para la construcción de un único sistema. CYOOA fue mejorada para validar y demostrar la efectividad de técnicas orientadas a objetos para soportar el análisis de dominio.

MDA:[5] Model-Driven Architecture. La arquitectura dirigida por modelos es un acercamiento al diseño de software, propuesto y patrocinado por el Object Managemente Group. MDA se ha concebido para dar soporte a la ingeniería dirigida a modelos de los sistemas software. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos.

---

[5] http://es.wikipedia.org/wiki/Model_Driven_Architecture

OMG:[6] Object Management Group. Grupo de gestión de objetos, es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas. El grupo está formado por compañías y organizaciones de software como: Hewlett-Packard (HP), IBM, Sun Microsystems, Apple Computer.

OOram: Object Oriented Role Analysis Method. Método de análisis de roles orientado a objetos es un método, basado en el concepto de rol, para ejecutar modelado orientado a objetos. OOram es el precursor del Lenguaje Unificado de Modelado (UML). OOram fue originalmente desarrollado por Trygve Reenskaug (1996), un profesor de la Universidad de Oslo y el fundador de la compañía noruega de informática Taskon.

PADDLE: Es un sistema transformacional que guarda una historia de desarrollo como una secuencia de transformaciones aplicadas.

PHP:[7] PHP es un lenguaje de programación interpretado, diseñado originalmente para la creación de páginas web dinámicas. Es usado principalmente en interpretación del lado del servidor pero actualmente puede ser utilizado desde una interfaz de línea de comandos o en la creación de otros tipos de programas incluyendo aplicaciones con interfaz gráfica usando las bibliotecas

[6] http://es.wikipedia.org/wiki/Object_Management_Group
[7] http://es.wikipedia.org/wiki/PHP

Qt o GTK+. PHP es un acrónimo recursivo que significa PHP Hypertext Pre-processor (inicialmente PHP Tools, o, Personal Home Page Tools).

REBOOT: REuse Based on Object-Oriented Techniques. Reutilización basada en técnicas orientadas a objetos, es proyecto cuyos objetivos son estudiar, desarrollar y diseminar metodologías avanzadas para desarrollo de software dirigidas a reutilización y orientadas a objetos con énfasis en reutilización planificada. El proyecto ha desarrollado dos modelos (desarrollo para reutilización y desarrollo con reutilización) que incorporan actividades específicas de reutilización en todas las fases del ciclo de vida de desarrollo, y un conjunto de herramientas (en entorno REBOOT) que respalda estos dos modelos y ayuda a una organización a administrar una librería de componentes reutilizables. Estos modelos y el entorno son fuertemente dependientes de la definición que se tenga de componente reutilizable.

RSEB: Reuse-Driven Software Engineering Business. Negocio de ingeniería de software dirigido por reutilización, es un proceso de reutilización sistemático dirigido por casos de uso: la arquitectura y los sistemas reutilizables son primero descritos por medio de casos de uso y entonces transformados en modelos de objetos que son fácilmente rastreables a estos caso de uso.

SETL: Lenguaje SET, es un lenguaje de programación de muy alto nivel basada en la teoría matemática de conjuntos. Originalmente fue desarrollada por Jack Schwartsz en el Instituto Courant de

Ciencias Matemáticas de la NYU a finales de los sesentas. SETL provee dos tipos de datos básicos agregados: conjuntos no ordenados y secuencias (posteriormente también llamados tuplas). Los elementos de conjuntos y tuplas pueden ser de cualquier tipo arbitrario, incluyendo conjuntos y tuplas en sí.

UML:[8] Unified Modeling Language. El lenguaje unificado de modelado es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocio y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes reutilizables. Es importante resaltar que UML es un "lenguaje" para especificar y no para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.

VHLL: Very High-Level Programming Language. Lenguaje de programación de muy alto nivel, es un lenguaje con un muy alto nivel de abstracción usado de forma primaria como una herramienta de productividad de programador profesional. Estos lenguajes usualmente están limitados a una aplicación muy específica, propósito, o tipo de tarea. Debido a esta limitación de

[8] http://es.wikipedia.org/wiki/Lenguaje_Unificado_de_Modelado

campo, es posible que utilicen sintaxis que nunca es utilizada en otros lenguajes de programación, tal como el la sintaxis directa en inglés. Por esta razón, los VHLL son a menudo referidos como lenguajes de programación orientados a objetivos.

RESUMEN

La reutilización de software es el proceso de implementar o actualizar sistemas de software utilizando activos del mismo. Aunque al principio podría pensarse que un activo de software es simplemente otro término para código fuente, éste no es el caso. Los activos de software o componentes incluyen todos los productos derivados del mismo, desde requerimientos y propuestas, especificaciones y diseños a manuales y juegos de prueba. Cualquier cosa que sea producto de un esfuerzo de desarrollo de software potencialmente puede ser reutilizada.

Un buen proceso de software facilita el incremento de la productividad, calidad y confiabilidad, y el decremento de los costos y tiempo de implementación. Una inversión inicial es requerida para empezar un proceso de reutilización de software, pero tal inversión se paga por sí sola en unas pocas reutilizaciones. A corto plazo, el desarrollo de un proceso de reutilización y repositorio produce una base de conocimiento que mejora en calidad después de cada reutilización, minimizando el monto de trabajo de desarrollo requerido para futuros proyectos y reduciendo el riesgo.

En este trabajo de graduación se presentan las ventajas de la reutilización, se desarrolla el concepto de los diferentes tipo de reutilización, la ingeniería de dominios aplicada a la reutilización de software, los principios del análisis y diseño orientado al objeto para la reutilización, las metodologías de reutilización de software. Para mostrar la reutilización con un ejemplo del método composicional de reutilización se definen ejemplos de patrones de diseño utilizado en php en diferentes escenarios de aplicación.

OBJETIVOS

General

1 Comparar los distintos tipos de metodologías de reutilización de software que actualmente se conocen y utilizan en proyectos informáticos.

2 Evaluar las ventajas y desventajas de cada uno de ellos para poder de esta manera tener un criterio que se pueda usar a la hora de aplicar alguna metodología específica.

Específicos:

1. Conocer distintas metodologías visualizando su implicación en la calidad, costo y productividad del software.

2. Conocer el análisis y diseño orientado al objeto para reutilización.

3. Evaluar los requisitos que son necesarios para llevar a cabo adecuadamente cada una de las clases de métodos de reutilización.

4. Evaluar la aplicación práctica de las metodologías de reutilización de software en proyectos informáticos, por medio de la aplicación de patrones para php.

INTRODUCCIÓN

Hoy en día se busca la producción de sistemas software de calidad de forma ágil, eficiente y sistemática. A menudo se dice que el desarrollo de software es una combinación de arte y ciencia por lo que se han dirigido muchos esfuerzos de investigación y desarrollo en la búsqueda de técnicas, metodologías, herramientas y procedimientos que ayuden a mejorar el desarrollo de dichos sistemas.

La reutilización de software es una excelente manera de ahorrar costos y esfuerzos de desarrollo y cuando se implementa cumpliendo los propósitos y correctamente, la reutilización puede ahorrar tiempo y dinero.

Los beneficios de la reutilización de componentes y activos de software han atraído la atención de un número considerable de empresas encargadas de desarrollar sistemas de software. Por este motivo, ha surgido recientemente un interés por crear y utilizar modelos de procesos para desarrollo de software que contemplen la reutilización. Aunque el concepto de reutilización no es nuevo, recientemente ha emergido, como elemento central de este, el concepto de componente de software reutilizable. Las características de los componentes de software reutilizables han obligado a crear nuevos modelos de desarrollo de software basados en la reutilización. Estos procesos buscan la solución a un problema. Esta solución no es única, es por esto que los métodos deben ir evolucionando y adaptándose a las nuevas necesidades de los diferentes tipos de usuarios y los nuevos conceptos de la Ingeniería de Software. Otra de las deficiencias encontradas en los métodos basados en la reutilización es que no especifican de forma detallada el ciclo de vida de un componente de software reutilizable. Estos métodos se centran en la reutilización del componente y no en su desarrollo individual.

En este trabajo se presentará una visión general de las metodologías de reutilización más utilizadas, los casos para los cuales son tomadas en cuenta, y una serie de ejemplos prácticos de aplicación de metodología composicional con patrones de diseño en lenguaje php.

1 MARCO TEÓRICO: CONCEPTOS BÁSICOS DE REUTILIZACIÓN

La reutilización es uno de los conceptos más simples y antiguos en la programación, y es algo que a menudo no es muy utilizado. Cuando se implementa cumpliendo los propósitos y correctamente, la reutilización puede ahorrar tiempo y dinero, a la vez que hace un inventario de activos de software reutilizables y valiosos.

1.1 Beneficios de la reutilización

1.1.1 Tiempo y costo

La reutilización de software es una excelente manera de ahorrar costos y esfuerzos de desarrollo. De forma ideal, el tiempo de desarrollo es reducido debido a que los componentes reutilizables relevantes pueden ser aplicables al proyecto dado en un marco de tiempo menor que redesarrollar desde cero. Como el tiempo y costo de desarrollo tienen una correlación positiva, reducir los tiempos de desarrollo trae un ahorro en los costos del proyecto. Esto puede llevar a proyectos y productos más baratos, y más utilidades a la organización.

El tiempo del producto al mercado puede ser reducido, dando una ventaja competitiva.

1.1.2 Calidad

Cualquier optimización, refactorización y pruebas hechas en componentes reutilizables ya han sido completados. Todas las lecciones aprendidas al producir el componente están implícitamente incluidas dentro de

esto y son automáticamente llevadas al siguiente proyecto. Consecuentemente, los proyectos desarrollados son de una mayor calidad.

Las organizaciones pueden esperar un incremento cuantitativo en la calidad del software. La reutilización efectiva puede reducir la densidad de defectos de 5 a 10 veces. Es más, debido a los ahorros en tiempo, los desarrolladores pueden invertir el mismo en el software nuevo que necesita ser producido, incrementando potencialmente su calidad o cualidades.

1.1.3 Estandarización

Los componentes dentro de un dominio dado (área de aplicación), requieren cierta clase de estandarización para hacerlos compatibles con otros componentes. Esto puede llevar a hacer estándares de interfaces de componentes, así como de código y documentación. Estas entidades animan a mejores prácticas de desarrollo.

1.2 Barreras a la reutilización

Con todos los beneficios de la reutilización de software efectiva, habría que preguntarse: ¿por qué la reutilización de software no es una práctica común en la ingeniería de software? La reutilización de software no es una idea nueva, así que no es cuestión de simplemente esperar para que la tecnología sea explotada. En cambio hay muchas razones clave por las cuales la reutilización no ha ganado una popularidad más amplia.

1.2.1 Elaboración, administración y uso de componentes reutilizables

La entidad esencial en la reutilización de software es el componente reutilizable. Pero hay muchas complejidades asociadas con el desarrollo de estos elementos. Los componentes tienen que ser genéricos en su naturaleza para ser reutilizados en múltiples sistemas (o múltiples momentos en el mismo sistema). Muchos componentes requerirán de alguna clase de generalización a través de la cual la mayoría de lógica específica de aplicación es removida y reemplazada con más funcionalidad de todo propósito.

¿Pero cuán general es demasiado general? Si un componente es hecho demasiado general, una larga porción de éste requerirá desarrollo cuando sea reutilizado, reduciendo de esta manera el beneficio de ahorro de tiempo de la reutilización. Por otro lado, si un componente es hecho demasiado específico, su reusabilidad puede reducirse considerablemente, anulando el propósito de la reutilización.

Además, los componentes de software necesitan ser capaces de trabajar con otros. Esto origina problemas con interfaces entre los mismos. Las interfaces necesitan ser estandarizadas entre componentes dentro de un dominio de aplicación dado.

1.2.2 Gastos al comenzar

Habrá un costo inicial en la organización por empezar a implementar una metodología de reutilización de software dentro del desarrollo. Los costos potenciales son las modificaciones a los procesos de desarrollo, empleo de personal para manejar la práctica, la generación y administración de un

repositorio de componentes. La gerencia necesitaría ser convencida que cualquier gasto será justificado en el camino.

1.2.3 Reto a los métodos tradicionales de ingeniería de software

Convencionalmente, el software ha sido construido de requerimientos desde cero. Sin embargo, la reutilización de software propone un cambio de paradigma de la ingeniería de software. Generalmente, en lugar de ser desarrollado desde cero, el software es ensamblado desde activos reutilizables.

La razón por la que ésta es una barrera a la reutilización de software efectiva, es porque la idea se desvía grandemente de la norma, y puede encontrar resistencia técnica inmediata. El desarrollo a través de la reutilización de software no es en lo que la industria ha estado poniendo sus esfuerzos en perfeccionar y, consecuentemente, la asistencia general y herramientas que den soporte a la reutilización pueden ser escasas. Además, hay cuestiones personales donde la gente puede sentir que la idea es demasiado buena para ser verdad.

Adicionalmente, se requieren cambios en la manera en que los desarrolladores se enfocan en los proyectos. Esto puede no ser de la satisfacción de aquellos quienes tienen sentimientos negativos por la reutilización de software, o quienes están contra cambios extremos en las prácticas de desarrollo.

1.3 Reutilización práctica de software

Hay muchas metodologías para poner en práctica la reutilización de software. Desde la reutilización de código ad hoc, a prácticas de reutilización

repetible, donde la reutilización es dentro de un proyecto o en múltiples proyectos.

Éstos pueden ser resumidos dentro de dos categorías generales:

Prácticas de reutilización con desarrollo
Prácticas de reutilización sin desarrollo.

1.3.1 Prácticas de desarrollo con reutilización

La reutilización de software es más comúnmente referida dentro del contexto de reutilización durante el desarrollo. Hay muchos enfoques hacia la reutilización de software durante el desarrollo. Entre ellos:

1.3.1.1 Tecnologías orientadas a objetos

Un concepto clave en las tecnologías de objetos es la idea de "objeto". Los objetos caracterizan el dominio del problema, cada uno teniendo atributos y comportamientos específicos. Los objetos son manipulados con una colección de funciones y se comunican unas con otras utilizando un protocolo de mensajes, y están organizados dentro de clases y subclases.

Un objeto encapsula la data y el proceso que es aplicado a la data. Esta importante característica hace posible clases de objetos a ser construidas e inherentemente llevan a librerías de clases reutilizables y objetos. Además, las tecnologías orientadas a objetos también brindan herencia. La herencia significa que un objeto puede heredar uno o más atributos de otros objetos en su jerarquía. De esta manera reutilizar objetos a través de adaptación. La capacidad de reutilizar objetos más allá de un simple sistema, significa que la

reutilización orientada a objetos es posible para reutilización interna en un proyecto, así como para reutilización a través de múltiples proyectos.

Ejemplos de tecnologías de objetos comerciales establecidas son:

- OMG/ CORBA
- Microsoft COM
- Componentes Sun JavaBean

Tecnologías de componentes emergentes incluyen:

- Marcos de Trabajo Microsoft .NET
- Sun Enterprise JavaBeans (EJB)
- Corba Component Model (CCM)

Pocas organizaciones han adoptado métodos orientados a objetos como su metodología primaria. Esto, debido a que la escalabilidad hizo estas tecnologías indeseables para reutilización. Sin embargo, los objetos permiten componentes a gran escala en lugar de solamente funciones para reutilización. De esta manera, mientras un lenguaje de programación orientado a objetos puede ser parte de las capacidades de reutilización de una organización, es sólo un elemento de un programa más grande y no debería ser utilizado por sí sólo.

1.3.1.2 Desarrollando con reutilización, reutilización oportunista

En una organización de desarrollo que practica reutilización oportunista, ésta es ad hoc, cuando la oportunidad para reutilización se presenta por sí

misma, es explotada. Por ejemplo, si una organización estuviera construyendo un sistema, y encuentra que durante el diseño o desarrollo, se podría ahorrar tiempo debido a que uno de sus subsistemas puede heredar las propiedades de otro, entonces esa organización está practicando reutilización oportunista.

Queda bastante claro que tal clase de desarrollo está bastante diseminada, por lo que muchas organizaciones están practicando inadvertidamente una forma de reutilización de software, aunque a través del uso de tecnologías orientadas a objetos, o a través de buenas prácticas de desarrollo.

En términos de cambio organizacional, no es necesario mayor trabajo para implementar la reutilización oportunista, debido a que recae en previsión y coincidencia. Esta clase de reutilización no afecta los procesos de desarrollo, así que es bastante barato de implementar dentro de una organización. El lado negativo, es que la reutilización oportunista no permitirá que la capacidad de reutilización sea predeciblemente repetible debido a su naturaleza ad hoc.

1.3.1.3 Desarrollando con reutilización: reutilización sistemática

La reutilización sistemática está “enfocada al dominio”. Se basa en un proceso repetible, y de manera primaria relacionada con la reutilización de los artefactos del ciclo de vida de más alto nivel, tales como: requerimientos, diseños y subsistemas. La idea detrás de la reutilización sistemática de software es que la reutilización no debería ser ad hoc, sino debería ser implementada dentro de la organización como parte del desarrollo de procesos. Contrastando directamente con la reutilización oportunista donde la reutilización no es parte del proceso.

Un concepto clave de reutilización sistemática es el “dominio”. Un dominio se refiere a un conjunto de sistemas cuyas propiedades comunes lo hacen ventajoso para estudiarlo y organizar estas propiedades dentro de una colección de componentes reutilizables, los cuales pueden ser entonces utilizados para construir sistemas en el dominio.

1.3.2 Prácticas de reutilización sin desarrollo con reutilización del producto

Hay otras clases de reutilización en relación con el desarrollo de software. Uno de estos tipos es la reutilización del producto, la que se refiere a la reutilización de un sistema entero. Esencialmente, en vez de construir un sistema específico para un sólo cliente, se construye un sistema más general que puede ser utilizado por muchos clientes. El ejemplo más obvio de reutilización de producto son los sistemas operativos para PC de escritorio en el mundo, tales como los productos Windows de Microsoft. En este caso, el software está construido para que sea capaz de ejecutarse sobre múltiples configuraciones de hardware. Sustituyendo el desarrollo de un sistema operativo para cada cliente. Esencialmente, el software es de hecho reutilizado a través de su implementación genérica.

2 INGENIERÍA DE DOMINIOS

Existen diversas definiciones para ingeniería de dominios, las cuales tienen como denominador común, el propósito de la ingeniería de dominios: proveer reutilización entre aplicaciones diferentes (una familia de aplicaciones). La ingeniería de dominios es un proceso automático para proveer una arquitectura esencial común para estas aplicaciones. Esto puede ser aplicado a sistemas existentes y a sistemas nuevos. De esta manera, la ingeniería de dominios abarca el análisis de aplicaciones existentes o de los conceptos de un área de problema.

2.1 Dominio

2.1.1 Definición de dominio

Un dominio puede ser definido como un conjunto de problemas o funciones que las aplicaciones de tal dominio pueden resolver. El término "dominio" puede ser utilizado asociándolo de diversas maneras:

- Área de negocios
- Colección de problemas (dominio de problemas)
- Colección de aplicaciones (dominio de soluciones)
- Área de conocimiento con terminología común

Para determinar un dominio existen varios criterios. La vista puede ser enfocada en describir lo que está dentro del dominio, cuáles son los límites del dominio, o qué está afuera del dominio.

El primer caso, describe los elementos que constituyen el dominio, o bien, identifica otros dominios, que juntos forman el dominio actual (los dominios pueden tener subdominios). El segundo caso, describe las reglas de inclusión y exclusión. El tercer caso, como también el primero, pueden ser representados por estructura y diagramas de contexto.

Sin embargo, los límites del dominio no son necesariamente fáciles de encontrar. Esto puede ser debido a la inexperiencia del analista de dominios y a los dominios que usa, o a los dominios que proveen servicios a otros dominios.

Los dominios pueden ser considerados al menos desde dos puntos de vista:

- Primero, como se consideró en un contexto orientado a objetos, un dominio puede ser visto como mundo real. Este enfoque se concentra en el fenómeno y sus procesos. Los requerimientos de diferentes aplicaciones no son tomadas en cuenta. De manera que esta clase de ingeniería de dominios es bastante similar al modelado conceptual, y cubre los primeros dos puntos de la lista previa. El modelo de dominio, derivado de esta alternativa, es altamente reutilizable, porque los conceptos de un área de problema son bastantes estables. Éstos no varían necesariamente, aunque los requerimientos de los sistemas cambian.

- El otro modo de considerar un dominio, es verlo como un conjunto de sistemas. Este punto de vista se concentra en familia de aplicaciones. El dominio es delimitado de acuerdo a las similitudes entre las aplicaciones. De este modo, este enfoque cubre el tercer punto de la lista previa. Esta última alternativa es más cercana a la ingeniería de

línea de producto, mientras que la anterior, se ajusta también a ingeniería de un sólo sistema.

2.1.2 Relación entre dominios y aplicaciones

Las aplicaciones pueden pertenecer a varios dominios. Por ejemplo, una aplicación concerniente a prácticas bancarias distribuidas, cubre al menos los siguientes dominios:

Prácticas bancarias, sistemas de información comerciales de bancos, administración de flujos de trabajo, interfaces de usuario, sistemas de administración de bases de datos y administración de red. Además, las aplicaciones no necesariamente cubren un dominio completo o varios dominios enteros. Por el contrario, varios dominios pueden ser dispersos bajo una aplicación.

Si la funcionalidad del dominio es cubierta por un sólo subsistema en un sistema, el dominio es llamado *encapsulado*. Por el contrario, si la funcionalidad del dominio está dispersa a través de varios subsistemas de un sistema, ese dominio se dice que es *distribuido*. Los dominios distribuidos también pueden ser llamadas *difusos*.

2.1.3 Dominios verticales y horizontales

Los dominios pueden ser divididos en verticales y horizontales. En los *dominios verticales*, los sistemas de software son clasificados de acuerdo al área de negocios. Tales sistemas son, por ejemplo, sistemas de reservación de aerolíneas, sistemas de registros médicos, sistemas de administración de portafolio, procesamiento de órdenes, y administración de inventarios.

En los *dominios horizontales*, partes de un sistema de software son clasificadas de acuerdo a su funcionalidad. Ejemplos son: sistemas de bases de datos, librerías contenedoras, sistemas de flujos de trabajo, librerías GUI (interfaces de gráficas de usuario) y librerías de código numéricas.

Los dominios verticales son dominios de sistemas, mientras que los dominios horizontales son dominios de partes de sistemas. Cuando se aplica la ingeniería de dominios a un dominio vertical, el resultado puede ser software reutilizable, el que se puede presentar como una forma de marco de trabajo y componentes reutilizables. En el caso de sistemas horizontales, la ingeniería de dominios puede producir componentes reutilizables.

Como en el caso de dominios distribuidos, también los dominios horizontales son encontrados más típicamente en sistemas heredados. Sin embargo, sistemas grandes o complejos puede estar dispersos sobre varios dominios, aún desde el principio. Los *dominios laterales* son dominios horizontales en un nivel de subsistema, y de esta forma, también pueden ser llamados *dominios de componentes*. Éstos corresponden a un conjunto de componentes fuertemente relacionados (o activos bases), que pueden pertenecer a varios sistemas como sus subsistemas.

2.2 Ingeniería de dominios e ingeniería de aplicación

Para hacer posible la ingeniería de línea de productos, una convención bien aceptada es dividir el proceso de ingeniería en dos diferentes procesos: ingeniería de dominios e ingeniería de aplicación. Estos procesos pueden ser seguidos en paralelo.

A la ingeniería de dominios también se le conoce como ingeniería *para* reutilización; y la de aplicación puede ser llamada ingeniería *con* reutilización. El propósito de la ingeniería de dominios es proveer los activos reutilizables esenciales que son explotados durante la ingeniería de aplicación cuando se ensamblan o personalizan aplicaciones individuales. De esta forma, estas fases son también llamadas *ingeniería de activos* e *ingeniería de productos*, o *desarrollo de activos esenciales* y *desarrollo de producto,* respectivamente.

Los subprocesos de ingeniería de dominios y la ingeniería de aplicación corresponden uno con otro. Las fases de ingeniería de dominios se concentran en el diseño e implementación de la arquitectura principal. Las fases de ingeniería de aplicación son llamadas *ingeniería de requerimientos*, *diseño análisis* e *integración y pruebas*.

La ingeniería de dominios puede también ser comparada con la ingeniería de software convencional. El *análisis de requerimientos* corresponde al análisis de dominio de tal manera que el análisis de requerimientos produce requerimientos para un sólo sistema, mientras que el análisis de dominios produce requerimientos reutilizables configurables para una familia de sistemas.

De forma similar, el *diseño de sistemas* busca el diseño de un sólo sistema; mientras que el diseño de dominio, se concentra en el diseño reutilizable para una clase de sistemas y un plan de producción. Finalmente, la *implementación de sistemas* despliega únicamente un sistema, mientras la implementación del dominio produce componentes reutilizables, arquitectura esencial, y procesos de producción.

2.3 Análisis del dominio, diseño del dominio e implementación del dominio

La ingeniería de dominios frecuentemente es dividida en tres fases: análisis de dominio, diseño de dominio, e implementación de dominio. Sin embargo, algunas referencias, por ejemplo, dividen la ingeniería de dominios en dos partes llamadas análisis de dominios e implementación de dominios. En este caso, el análisis de dominios cubre también tareas del diseño de dominios. En la literatura, los términos concernientes a las fases han sido utilizados inconsistentemente. Por ejemplo, la ingeniería de dominios y el análisis de dominios han sido utilizados intercambiándolos. No obstante, más recientemente el análisis de dominio es establecido como parte de ingeniería de dominios.

2.3.1 Análisis del dominio

El concepto de análisis de dominio se utiliza para denotar el dominio de problema de una familia de aplicaciones. El término correspondiente para sistemas simples es análisis de sistemas. El análisis de dominio está asociado con la reutilización. Su propósito es capturar la información involucrada con el dominio a ser reutilizado en desarrollar aplicaciones futuras en el mismo dominio. Sin embargo, el análisis de dominios no está necesariamente restringido a la ingeniería de línea de producto; en cambio, puede ser utilizada en el contexto de ingeniería de un sólo sistema también.

La salida de un análisis de dominio es el *modelo de dominio.* Tratando de encontrar los elementos más comunes de varios modelos de dominios, los ingredientes del modelo de dominio puede ser esbozado como sigue:

- Alcance del dominio (definición de dominio, análisis de contexto)
- Análisis de elementos comunes
- Diccionario de dominio (léxico de dominio)
- Notaciones (modelado de concepto, representación de concepto)
- Ingeniería de requerimientos (modelado de características)

En la lista de arriba el primer término después de cada inciso denota el término utilizado en este trabajo, mientras el paréntesis encierra los términos alternativos para el mismo concepto. Una breve descripción es dada para término de la siguiente manera:

Alcance del dominio busca los límites del dominio. Esta actividad da ejemplos de las aplicaciones pertenecientes al dominio; así como ejemplos de tales aplicaciones que están afuera del dominio. Además, da las reglas de exclusión e inclusión.

Nótese que el alcance puede ser dividido en alcance del dominio, alcance de línea de producto, y alcance de activos. El alcance de dominios está asociado con la ingeniería de dominios. El *alcance de línea de producto* identifica los requerimientos y productos que deberían pertenecer a cierta línea de productos. El *alcance de activos* identifica los activos reutilizables esenciales.

Análisis de elementos comunes considera los elementos comunes y variaciones de la aplicación en el dominio. Estudia los requerimientos y propiedades de las aplicaciones y los conceptos en el dominio. El análisis de elementos comunes produce un *documento de elementos*

.

Tabla I. Documento de elementos

Visión general	Descripción del dominio y su relación con otros dominios
Definiciones	Un conjunto estándar de términos técnicos
Elementos comunes	Una lista estructurada de suposiciones que son verdaderas para cada miembro de la familia
Variaciones	Una lista estructura de suposiciones acerca de cómo los miembros de la familia difieren
Parámetros de variación	Una lista de parámetros que refine las variabilidades, añadiendo un rango de valores y vinculando tiempo a cada uno
Asuntos	Un registro de decisiones y alternativas importantes.
Escenarios	Ejemplos usados en describir elementos comunes y variabilidades

Diccionario de dominio provee y define los términos concernientes al dominio. Su propósito es establecer comunicación entre los desarrolladores y otros participantes del proyecto de forma más fácil y precisa. El diccionario de dominio puede ser parte del documento de elementos comunes.

Notaciones proveen una manera uniforme de representar los conceptos utilizados en el modelado de dominio. Ejemplos de tales notaciones son diagramas de objetos, diagramas de transición de estados, entidad-relación, y diagramas de flujo. Es razonable utilizar tal notación que es familiar para la mayoría de los participantes del proyecto.

Algunas notaciones se supone que describan un sistema, y por lo tanto, pueden ser llamadas notaciones de *nivel de sistema*. No es fácil o aún deseado,

utilizar estas notaciones para describir los conceptos de *nivel de dominio*. Por ejemplo, UML (lenguaje unificado de modelado) puede ser utilizado para describir variabilidades dentro de un sólo sistema. Sin embargo, de acuerdo a algunos autores, podría ser confuso de utilizar (o expandir) las mismas notaciones para describir también variabilidades entre sistemas. De esta manera las notaciones familiares y nuevas tienen sus ventajas y desventajas. Por un lado, las notaciones familiares y sus extensiones son fáciles de aprender; por otro lado, pueden conducir a malentendidos si son utilizados en situaciones inapropiadas o niveles abstractos. Escoger una notación es una importante decisión porque éstas tienen un rol clave en proveer comprensión de un sistema o un dominio.

Ingeniería de requerimientos conlleva reunir, definir, documentar, verificar, y administrar los requerimientos que especifican las aplicaciones en el dominio. En el contexto de línea de producto, el propósito es reutilizar y configurar los requerimientos entre aplicaciones individuales. Tales requerimientos pueden también ser llamadas características. Una característica puede también ser definida como una cualidad del sistema que es visible al usuario final, o también puede involucrar cualidades que son relevantes a algunos participantes del proyecto (no necesariamente visibles).

Las características pueden ser representadas por modelos que también dicen cuáles combinaciones de éstas son significativas. Los modelos de características proveen notaciones para diferentes clases de características tales como las que son parecidas al FODA, es decir, características mandatarias, opcionales y alternativas. Sin embargo, estos tres tipos de características no son siempre adecuadas. Por ejemplo, de características alternativas se puede seleccionar exactamente una. En algunas situaciones, debería ser posible escoger cualquier número de características de un conjunto

dado. Además de diferentes clases de características, los modelos pueden dar una forma de presentar restricciones entre características o bases lógicas para seleccionarlas.

Además de la lista dada de tareas concernientes al modelo de dominios, el análisis de dominio comprende la reunión de información de diferentes fuentes tales como diversos documentos expertos de dominio. Además, puede cubrir clusterización, abstracción, clasificación, y generalización de conceptos de dominios. El análisis de dominio es diferente para aplicaciones diferentes y para dominios diferentes. Por ejemplo, sistemas interactivos pueden requerir casos de uso y escenarios. Así mismo, la aplicación centrada en datos puede ser más fácilmente descrita por diagramas entidad-relación o diagramas de objetos. Propiedades especiales tales como soporte en tiempo real, distribución y tolerancia a fallos pueden requerir técnicas especiales de modelado.

Propiedades peculiares a cierta organización pueden tener requerimientos adicionales. Es importante notar que, el análisis de dominio puede también ser llamado *modelado de dominio* referente a la salida del modelo de dominio. Adicionalmente, hay otras inconsistencias en la terminología y en los contenidos de las subáreas concernientes al análisis de dominio. En este trabajo que usa la división más común de subáreas, el alcance de dominio es una parte del análisis de dominio. Sin embargo, el alcance de dominio y análisis de dominio pueden ser representados como partes de la ingeniería de dominios. Además, se seguirá el estudio de la propuesta que sigue la teoría que el alcance de dominio y la ingeniería de requerimientos son partes del análisis de dominio. Sin embargo, el alcance de dominio puede ser considerado como parte de la ingeniería de requerimientos, que en cambio, pertenece al análisis de dominio.

2.3.2 Diseño del dominio

Diseño de dominio significa perfilar la arquitectura esencial para una familia de aplicaciones. Esto abarca la selección del estilo de arquitectura. Así, la arquitectura común bajo el diseño debería ser representada usando diferentes vistas. La arquitectura esencial debería también proveer variabilidad entre aplicaciones. En esta fase, se decide cómo habilitar esta variabilidad o configurabilidad. De acuerdo a los modelos de características y documentos de elementos comunes, también se debería seleccionar cuales componentes o elementos (tales como requerimientos), son datos en la arquitectura principal y cuáles elementos son implementados como variaciones de aplicaciones individuales.

El diseño de dominio produce también un *plan de producción* que dice cómo la aplicación concreta puede ser derivada de la arquitectura principal y desde los componentes reutilizables. El plan de producción comprende las descripciones de los sistemas con sus interfases para cada cliente, directrices para el proceso de ensamblaje de los componentes, y directrices para administrar las solicitudes de cambios de los clientes. El proceso de ensamblaje puede ser manual, semi-automático, o automático.

El diseño de dominio puede estar envuelto en evaluar la arquitectura esencial, analizar la misma contra sus requerimientos de calidad para revelar riesgos potenciales concernientes a la arquitectura. Hay diferentes opiniones del momento más adecuado para evaluar una arquitectura. Una arquitectura debería ser evaluada en una etapa temprana del desarrollo de una arquitectura de línea de producto, debido a que mientras más pronto los problemas son detectados más fáciles son de resolver. De esta manera, tan pronto como los

artefactos son evaluados, debería ser verificado cuán bien la arquitectura cumple los requerimientos.

La composición y otras acciones para proveer aplicaciones finales pueden requerir herramientas especiales que son destacadas en la fase de diseño. Un entorno constituido de tales herramientas puede ser llamado un *entorno de ingeniería de* aplicación.

2.3.3 Implementación del dominio

La implementación del dominio cubre la implementación de la arquitectura, componentes, y herramientas diseñadas en la fase previa. Ésta abarca, por ejemplo, escribir documentación e implementar lenguajes y generadores específicos de dominio. El propósito de la ingeniería de dominios es producir activos reutilizables que son implementados en esta fase. De esta manera, el resultado de toda la fase de ingeniería de dominios comprende componentes, modelos de características, modelos de análisis y diseño, patrones, marcos de trabajo, lenguajes específicos de dominio, planes de producción y generadores.

Es relevante hacer notar que la ingeniería de dominios es un proceso continuo. El conocimiento concerniente al dominio debería ser mantenido y actualizado todo el tiempo de acuerdo a la nueva experiencia, alcance, ensanchamiento, y nuevas tendencias. Además, la ingeniería de dominios debería adaptarse de acuerdo a la retroalimentación de la ingeniería de aplicación. De esta manera, el modelo de dominio nunca puede ser completado, podría siempre ser refinado para ser más exacto. Además, el modelo de dominio usualmente contiene alguna clase de compromiso acerca de diferentes y quizás inconsistentes puntos de vista de varios expertos.

2.4 Distinción de términos relacionados

2.4.1 Línea de producto

Los términos “dominio” y “línea de producto” son muy cercanos uno con otro. Sin embargo, la diferencia es que un dominio consiste en elementos conceptuales, mientras una línea de producto comprende productos concretos o aplicaciones a ser desarrolladas. Es más, estos términos pueden ser considerados para referirse al mismo nivel de abstracción, pero términos diferentes son pretendidos para audiencia diferente. Personal técnico (desarrolladores de software) entienden y usan el término “dominio”, mientras que para administradores, es más fácil hablar acerca de líneas de producto, porque es más parecido a términos de negocios o mercadeo.

2.4.2 Ingeniería de requerimientos

La ingeniería de requerimientos puede ser dividida en subtareas tales como *licitación de requerimientos* para descubrir y entender las necesidades del usuario; *análisis de requerimientos* para refinar las necesidades del usuario; *verificación de requerimientos* para asegurarse que los requerimientos del sistema están completos, correctos y consistentes; y *administración de requerimientos* para calendarizar y coordinar las actividades anteriores concernientes a requerimientos. Sin embargo, como en el análisis de dominio e ingeniería de dominios, también el análisis de requerimientos y la ingeniería de requerimientos son a veces utilizados intercambiándolos.

Nótese que el análisis de requerimientos para un sistema simple corresponde al análisis de dominio para una familia de sistemas. Complementariamente, la ingeniería de requerimientos es una parte de

documento de elementos comunes, y tiene una cercana conexión a las características.

Aunque el análisis de requerimientos puede ser considerado como análisis de dominios para sistemas simples, también está en relación con familias de sistemas. En la ingeniería de línea de productos, los requerimientos de ingeniería concernientes a toda la línea de producto deberían ser manejados separadamente de los requerimientos individuales de cada sistema. De esta manera, las arquitecturas de línea de producto consideran los requerimientos desde el punto de vista de productos de familias y sistemas simples. De hecho, aplicar la ingeniería a los requerimientos comunes define los elementos comunes, mientras que los requerimientos individuales se concentran en las variabilidades.

Esta es la explicación de por qué la ingeniería de requerimientos está conectada por un lado, con el análisis de elementos comunes; por el otro, significa mucho, tanto para la ingeniería de dominios como para sistemas simples.

2.4.3 Ingeniería de reutilización

La *ingeniería de reutilización* significa la identificación de partes reutilizables en el software y cambiar el software para hacerlo más reutilizable. Tareas involucradas en la ingeniería de reutilización son: recuperación de objetos, identificación de tipos de datos abstractos y componentes reutilizables, y la construcción de librerías de los componentes identificados. De esta forma, el aspecto de reutilización en la ingeniería de reutilización está concentrado en software existente. Para enfatizar aún más el aspecto heredado, se puede usar para el mismo propósito, un término muy cercano, *reingeniería de reutilización* y

extraer componentes reutilizables de software existente y colectarlo en repositorios.

Ambos términos (ingeniería de la reutilización y reingeniería de la reutilización) están asociados con promover la reutilización del software existente. La ingeniería de dominios en cambio, trata con el software existente y con el nuevo desarrollado. De esta manera, la ingeniería de reutilización puede ser considerada como parte de la ingeniería de dominio.

2.4.4 Arquitecturas de software específicas de dominio

La ingeniería de dominios es el proceso de desarrollar e implementar la arquitectura de software específica de dominio. *Una arquitectura de software específica de dominio* (DSSA por sus siglas en inglés) es una arquitectura para un dominio específico, sin embargo, debería ser todavía general para soportar varias aplicaciones del dominio. Un DSSA consiste de requerimientos comunes y el proceso para refinarlo. De este modo, un DSSA denota en mucho, lo mismo que una arquitectura de línea de producto.

2.4.5 Arquitecturas de referencia

Una *arquitectura de referencia* es una arquitectura de software para una familia de sistemas de aplicación. El resultado de refinar o instanciar una arquitectura de referencia es una *arquitectura de aplicación*. De esta manera, una arquitectura de referencia corresponde a una arquitectura esencial común; y una arquitectura de aplicación corresponde a la arquitectura de una aplicación individual en terminología de línea de producto.

Cuando se comparan DSSA y arquitecturas de referencia, los DSSAs de hecho, se concentran en el proceso en cómo proveer las características comunes y variables, y cómo derivar las arquitecturas finales para cada aplicación. Las arquitecturas de referencia y arquitecturas de aplicación, en cambio, denotan las arquitecturas y artefactos actuales. Sin embargo, las DSSAs y las arquitecturas de referencia tienen una cercana conexión una con la otra. Un DSSA es un proceso para desarrollar una arquitectura de referencia para generar aplicaciones dentro de un domino particular.

2.5 Métodos de ingeniería de dominios

La mayoría de los métodos se han concentrado originalmente en ingeniería de dominios. Más tarde, han sido extendidos o conectados a otros métodos para cubrir todo el proceso de ingeniería de línea de producto.

FODA (análisis de dominio orientado a las características), considera las características de aplicaciones similares en un dominio. Las características son capacidades de las aplicaciones consideradas desde el punto de vista del usuario final. Éstas cubren aspectos comunes y variables entre sistemas relacionados. Están divididas en mandatarias (o comunes), alternativas y características opcionales. FODA incluyen diferentes análisis tales como análisis de requerimientos y análisis de características. Esto produce un modelo de dominio cubriendo las diferencias entre aplicaciones relacionadas. FODA es actualmente, una parte del enfoque basado en modelo de la ingeniería de dominios. Este enfoque cubre la ingeniería de dominios y la ingeniería de aplicación. La parte de ingeniería de dominios consiste en análisis de dominio, diseño de dominio, e implementación de dominio.

Adicionalmente, FODA está más extendido dentro de FORM (método de reutilización orientado a la característica por sus siglas en inglés), para incluir también diseño de software y fases de implementación. FORM cubre el análisis de características de dominio y utiliza estas características para desarrollar artefactos reutilizables de dominio.

La ODM (organización de modelado de dominio) principalmente, se concentra en la ingeniería de dominio sistemas heredados. Sin embargo, puede ser aplicado a los requerimientos para nuevos sistemas. ODM es ajustable y configurable, y puede ser integrado con otras tecnologías de ingeniería de software. Esto combina diferentes artefactos tales como requerimientos, diseño, código, y procesos de varios sistemas heredados dentro de activos reutilizables comunes. ODM tiene soporte de DAGAR (Generación basada en arquitectura para reutilización en Ada, por sus siglas en inglés). El proceso DAGAR no cubre modelado de dominio. De esta manera, aplica ODM u otros métodos para este propósito. En lugar de esto, el proceso DAGAR incluye actividades para ingeniería de dominios e ingeniería de aplicación.

El negocio de ingeniería de software conducida por reutilización (Reuse-driven Software Engineering Business), RSEB es un método de reutilización sistemático y conducido por modelo. Se compone de conjuntos de aplicaciones relacionados desde conjuntos de componentes reutilizables. RSEB utiliza UML para especificar sistemas de aplicaciones, sistemas de componentes reutilizables y arquitecturas en capas. Las variabilidades entre sistemas están expresadas con puntos de variación y variantes añadidas. RSEB caracterizado, conecta características (desde FODA) con RSEB. De hecho, FODA y RSEB tienen mucho en común. Ambos son métodos conducidos por modelo proveyendo varios modelos correspondiendo a los diferentes puntos de vista del dominio.

De esta forma, son compatibles unos con otros. PuLSE (ingeniería de software de línea de producto, por sus siglas en ingles), divide el ciclo de vida de línea de producto en tres partes. En la fase de inicialización, PuLSE es personalizada para ajustarse a la aplicación particular. La adaptación es afectada por la naturaleza del dominio, la estructura del proyecto, el contexto organizacional, y los objetivos de la reutilización. En la segunda fase, la infraestructura de línea de producto es construida. Este paso incluye el alcance, modelado y arquitectura de la línea de producto. En la tercera fase, la infraestructura de línea de producto es utilizada para hacer productos individuales. Esto comprende instanciar el modelo de línea de producto y la arquitectura. Cada una de estas fases está asociada con la evolución de la infraestructura de línea de producto. Cada fase debería considerar cambiar requerimientos y cambiar conceptos dentro del domino. PuLSE tiene varios componentes. PuLSE-DSSA (PuLSE – arquitectura específica de dominio), por ejemplo, desarrolla una arquitectura específica de dominio basada en el modelo de línea de producto. Como otros ejemplos, PuLSE-Eco se concentra en alcance económico, y PuLSE-EM en evolución y administración.

El proceso FAST, abstracción, especificación y traducción orientada a la familia, por sus siglas en inglés, cubre el proceso de ingeniería de línea de producto completo. Sin embargo, divide la ingeniería de dominios en dos partes: análisis de dominios e implementación de dominio.

De esta forma, los problemas que involucran el diseño de dominio son considerados en la fase de análisis de dominio. FAST provee guía sistemática para cada paso durante la ingeniería de línea de producto. Estos pasos pueden ser seguidos como transiciones entre estados. FAST también describe cuáles artefactos son producidos en cada fase.

2.6 Ingeniería de dominios y metodologías orientadas al análisis y diseño orientados a objetos (OOA/D)

2.6.1 Defectos en los métodos OOA/D existentes

Los métodos de análisis y diseño orientado a Objetos (OOA/D), son ampliamente utilizados en la ingeniería de software. No obstante, éstos principalmente se concentran en análisis y diseño de sistemas simples. La orientación a objetos fue primeramente pensada para traer reutilización de forma inherente, con todo, hoy en día es reconocido que para poder hacer reutilización, se tiene que invertir y planear de antemano.

La reutilización está conectada a la orientación a objetos, al menos en la forma de marcos de trabajo y patrones diseño. Un marco de trabajo comprende un diseño abstracto del cual varias aplicaciones pueden ser instanciadas. Un patrón de diseño, en cambio provee una solución (diseño) para problemas recurrentes. Sin embargo, OOA/Ds no tienen soporte para reutilización, y no son guías en diseñar familias de aplicaciones. Los problemas de los métodos OOA/D han sido listados como sigue a continuación:

- No hay distinción entre ingeniería de dominios e ingeniería de aplicación.
- No hay fase de alcance de dominio.
- No hay diferenciación entre la variabilidad de modelado dentro de un aplicación y entre varias aplicaciones.
- Sin medios independientes de implementación de variabilidad de modelado.

Para resolver el problema de reutilización en OOA/Ds, hay varias propuestas. La primera alternativa es actualizar los métodos de ingeniería de dominios más antiguos. Por ejemplo, el análisis y diseño estructurado utilizado en FODA, puede ser reemplazado con métodos OOA/D. Segundo, los métodos de ingeniería de dominios personalizables pueden ser especializados en una dirección orientada a objetos. Por ejemplo, en ODM, un método de ingeniería de sistemas puede ser dado como un parámetro. Tercero, los métodos OOA/D pueden ser extendidos con los conceptos de ingeniería de dominios. Por ejemplo, RSEB, está basado en notación de modelado orientada a objetos. La cuarta alternativa es integrar métodos existentes. Por ejemplo, FeatuRSEB es desarrollado desde FODA y RSEB, la anterior ingeniería de dominios de aplicación y los conceptos posteriores orientados a objetos.

2.6.2 Metodologías OOA/D apoyando a la ingeniería de dominios

OOram (análisis de roles y modelado orientado a objetos, Object Oriented role analysis and modeling) es un método OOA/D que también enfatiza la ingeniería de dominios. OOram no cree en una sola metodología; en lugar de esto, es un método genérico que provee un marco de trabajo para crear una variedad de metodologías. Los sistemas de software reales son típicamente muy largos y complejos. De esta forma, OOram colecta objetos teniendo una meta común al mismo grupo llamado *colaboración*. Adicionalmente, diferentes puntos de vista son dados para entender el sistema más fácilmente.

Como los OOA/D se concentran en la dimensión de clase, OOram enfatiza la dimensión de modelo de rol. El *modelo de rol* colecciona objetos colaboradores juntos de acuerdo a su meta común. La meta define un *área de concernencia*. Un *rol* describe un objeto y su responsabilidad de lograr la meta

en el área de concernencia. Los modelos de roles pueden ser relacionados jerárquicamente entre ellos como un modelo de roles base y un modelo de roles derivado. Las colaboraciones son conectadas a marcos de trabajo y patrones de diseño. Un diseño de marco de trabajo puede ser representado como una composición de colaboraciones. Además, las colaboraciones pueden ser instancias de patrones de diseño.

OOram identifica los siguientes tres procesos para logar la meta dada. Primero, el *proceso de creación de modelo* se concentra en la creación de un modelo para un cierto fenómeno. Son ejemplos: crear un modelo de roles, ejecutar síntesis de modelo de roles, y crear especificaciones de objetos. Segundo, el *proceso de desarrollo de sistemas* cubre el típico ciclo de vida del software, desde la especificación de las necesidades del usuario hasta la instalación y mantenimiento del sistema. Tercero, el *proceso de construcción de activos reutilizables* es necesitado en la continua producción de varios sistemas cercanamente relacionados. Crear un sistema es principalmente configurar y reutilizar componentes robustos y probados. Puede ser necesario, para completar el sistema, añadir unos pocos nuevos componentes.

Además de OOram, JODA (análisis de dominio orientado a objetos JIAWG por sus siglas en inglés) - donde JIAWG (Joint Integrated Avionics Working Group) significa grupo de trabajo integrado de aviación por sus siglas en inglés – integra análisis de dominio con análisis orientado a objetos (OOA). El análisis de dominio provee un modelo de dominio para ser utilizado en producir elementos reutilizables especialmente requerimientos reutilizables; mientras OOA trae las técnicas de análisis de objetos y notaciones.

Los objetos son más estables que las funciones. Aunque la operabilidad de un objeto puede cambiar, el objeto en sí mismo permanece más a menudo.

Ésta es la razón por la que los métodos OOA/D son preferidos a los métodos funcionales.

JODA divide el proceso de análisis de dominio en tres fases:

- Preparación de dominio.
- Definición de dominio.
- Modelado de dominio.

El modelado de dominio es extendido desde OOA, y consiste de tres pasos. Primero, el historial de vida del objeto y la respuesta de estados-eventos son examinados. Este paso es derivado directamente desde el OOA, y comprende la identificación de objetos, definición de atributos y servicios, y encontrar las relaciones entre objetos. Segundo, escenarios de dominios son identificados, definidos y simulados. En esta fase, el comportamiento de los objetos es identificado para proveer servicios a los usuarios y otros sistemas. Tercero, los objetos son abstraídos y agrupados para habilitar la reutilización. Por ejemplo, si hay instancias repetidas de problemas similares tales como el procesamiento de muchos formatos de mensajes, los problemas son analizados para encontrar una abstracción para cubrir todos los casos. Así mismo, esta fase identifica elementos comunes y variabilidades tales como interfases estables con implementaciones variables. Por ejemplo, interfases de controladores de dispositivos para paquetes gráficos pueden ser similares, aunque las implementaciones varíen de acuerdo a las capacidades del dispositivo. Los tres pasos son iterados y el modelo de dominio resultante es revisado y actualizado tanto como el modelo es lo suficientemente exacto para definir el dominio.

Como un tercer ejemplo de métodos conectando orientación a objetos e ingeniería de dominios, es considerado FeatuRSEB. Como se mencionó

anteriormente FeatuRSEB es desarrollada desde FODA y RSEB. FeatuRSEB conecta casos de uso de RSEB y características de FODA entre ellos. El modelo de caso de uso es orientado al usuario, mientras que el modelo de características es orientado al reutilizador. El anterior provee una descripción de lo que los sistemas en el dominio hacen. El siguiente, en cambio, muestra cuál funcionalidad puede ser seleccionada cuando se aplica la ingeniería a nuevos sistemas en el dominio.

FeatuRSEB analiza varios sistemas ejemplares, encuentra sus elementos comunes y variabilidades, y produce un modelo de características. El proceso de construcción del modelo de características puede ser dividido en cuatro pasos:

- Primero, modelos de casos de uso individuales y ejemplares son fusionados dentro de un modelo de casos de uso de domino (o familia de modelos de caso de uso). Las diferencias son expresadas utilizando puntos de variación. El modelo es formado de tal manera que los ejemplares originales pueden ser rastreados.

- Segundo, un modelo de características inicial es creado. Las características funcionales son derivadas del modelo de casos de uso de dominio. Características mandatarias y opcionales son identificadas de acuerdo a su frecuencia de ocurrencia en sistemas ejemplares. Las características identificadas son descompuestas en subcaracterísticas. El modelo de casos de uso de dominio puede contener varios puntos de variación. Las características correspondientes a tales puntos de variación son divididas en subcaracterísticas. Las subcaracterísticas derivadas son rastreadas atrás hasta sus ocurrencias en el modelo de casos de uso de dominio.

- El tercer paso añade características arquitectónicas al modelo de características. En lugar de la función específica, las características arquitectónicas se relacionan a la estructura del sistema y configuración.

- Finalmente el cuarto paso añade características de implementación al modelo de características.

2.7 Problemas emergentes

Como conclusión se reúnen los problemas emergentes concernientes a la ingeniería de dominios. Sin embargo, los problemas listados no son necesariamente problemas de investigación:

Terminología no establecida

Es principalmente debida a la inmadurez del área de investigación.

Ingeniería de dominio incompleta

Significa que el modelado de dominio nunca puede ser completado; en cambio, el modelo puede ser siempre hecho más exacto. Además, el conocimiento del dominio de diferentes fuentes puede ser inconsistente entre ellos.

Dominios distribuidos

Están típicamente asociados con la ingeniería de dominios de aplicaciones existentes, es decir aspectos de reingeniería.

Dominios traslapados

Son debidos a las dificultades en determinar los límites del dominio o en dividir dominios en subdominios.

Falta de soporte para ingeniería de dominios en técnicas de modelado de objetos bien aceptadas

Es debido a la inmadurez del área de investigación de la ingeniería de dominios y a la creencia que la orientación a objetos inherentemente hace aplicaciones reutilizables.

El último problema es probablemente el más severo de los arriba mencionados. En adición, para las técnicas de modelado de objetos, las deficiencias similares conciernen al lenguaje de modelado UML. Este no tiene soporte ingeniería de dominios tampoco. Sin embargo, la técnica concerniente a MDA (Model-Driven Architecture, arquitectura conducida por modelo por sus siglas en inglés) que está basada en UML, puede proveer soporte a la ingeniería de dominios también. De esta forma, este tópico es un sujeto importante de futuras investigaciones.

3 PRINCIPIOS DEL ANÁLISIS Y DISEÑO ORIENTADO AL OBJETO PARA REUTILIZACIÓN

Todas las nociones que se refieren al diseño orientado al objeto se expresan en lo que Robert C. Martin llama “los principios del DOO”.
Estos principios son:

1. El principio de abierto/cerrado.
2. El principio de sustitución de Liskov.
3. El principio de inversión de dependencia.
4. El principio de separación de la interfaz.
5. El principio de equivalencia reutilización/publicación.
6. El principio de cierre común.
7. El principio de reutilización común.
8. El principio de dependencia acíclica.
9. El principio de las dependencias estables.
10. El principio de las abstracciones estables.

3.1. El principio de abierto/cerrado (PAC)

De todos los principios del diseño orientado al objeto, este es el más importante. Simplemente significa lo siguiente: se deberían escribir los módulos de tal manera que se puedan extender, sin tener que recurrir a ellos para ser modificados. En otras palabras, se desea que sean capaces de cambiar lo que hacen sin cambiar el código fuente de los mismos.

Esto podría sonar contradictorio, pero hay varias técnicas para lograr el PAC (principio abierto/cerrado) en larga escala. Una de éstas técnicas está basada

en la abstracción. De hecho la abstracción es la clave del PAC. A continuación algunas de estas prácticas son descritas.

3.1.1 Poliformismo dinámico

Considérese el siguiente código en C++:

```
struct Modem
{
  enum Type {Guatemala, Salvador, Honduras) type;
};
struct Guatemala
{
  Modem::Type type;
      // modem relacionado a Guatemala
};
struct Salvador
{
  Modem::Type type;
      // modem relacionado al Salvador
};
struct Honduras
{
  Modem::Type type;
      // modem relacionado a Honduras
};
void Logueo(Modem& m, string& pno, string& usuario, string& password)
{
  if (m.type == Modem::Guatemala)
```

```
    MarcaGuatemala((Guatemala&)m, pno);
  else if (m.type == Modem::Salvador)
    MarcaSalvador((Salvador&)m, pno);
  else if (m.type == Modem::Honduras)
    MarcaHonduras((Honduras&)m, pno)
}
```

En este ejemplo la función debe ser cambiada cada vez que un tipo de modem es añadido al software. Lo que es peor, ya que cada diferente tipo de modem depende de la enumeración Modem::Type, cada modem debe ser recompilado cada vez que un nuevo tipo de modem es añadido. Por supuesto, este no es el peor atributo de esta clase de diseño. Los programas que son diseñados de esta manera tienden a ser inundados de sentencias similares del tipo if/else or switch . Cada vez que cualquier cosa necesita ser hecha al modem, una sentencia if/else o switch se necesitará para seleccionar las funciones apropiadas a usar. Cuando nuevos modems son añadidos, o las políticas de los modems cambian, el código debe ser revisado para todas estas sentencias de selección, y cada uno debe ser modificado apropiadamente.

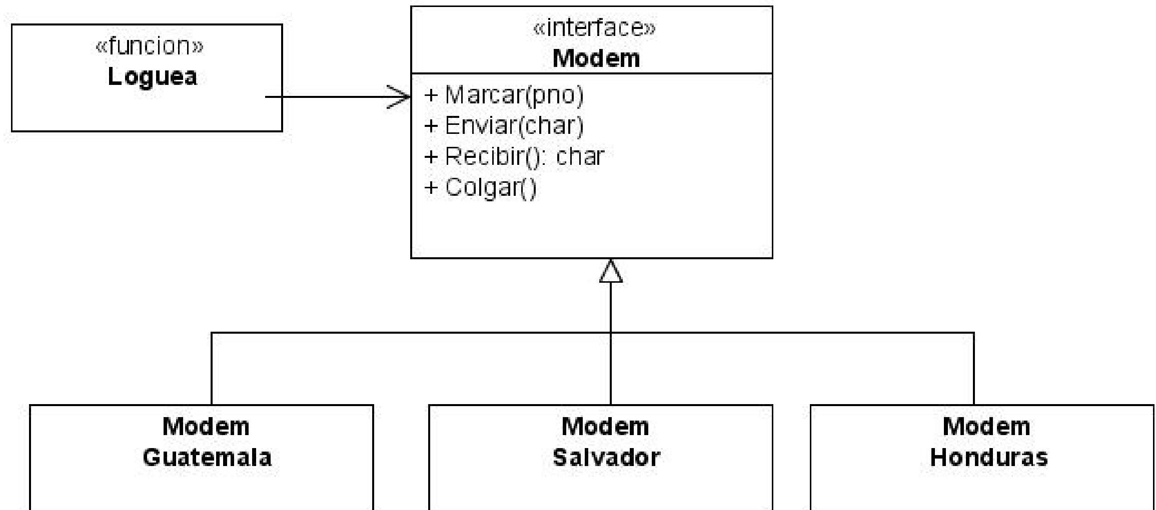
Lo que es peor, los programadores pueden utilizar optimizaciones locales que esconden la estructura de las sentencias de selección. Por ejemplo, podría ser que la función es exactamente la misma para los modems de Guatemala y Salvador. De esta manera se podría ver código como el siguiente:

```
if (modem.type == Modem::Guatemala)
        EnviaGuatemala((Guatemala&)modem, c);
else
        EnvíaSalvador((Salvador&)modem, c);
```

Claramente, tales estructuras hacen que el sistema sea mucho más difícil de mantener y son muy propensos al error. Como ejemplo del PAC considérese la figura 1. Aquí la función Loguea depende solo de la interface Modem. Un modem adicional no causará que la función Loguea cambie. De esta manera, se ha hecho un módulo que puede ser extendido con nuevos módulos sin requerir modificación, véase el siguiente código:

```
// Logueo ha sido cerrado para modificación
class Modem
{
  public:
    virtual void Marcar(const string& pno) = 0;
    virtual void Enviar(char) = 0;
    virtual char Recibir() = 0;
    virtual void Colgar() = 0;
};
void Logueo(Modem& m, string& pno, string& usuario, string& password)
{
  m.Marcar(pno);
}
```

Figura 1. Logueo cerrado para modificación



3.1.2. Polimorfismo estático

Otra técnica para estar conforme al PAC es a través del uso de plantillas genéricas. El siguiente código muestra cómo se hace esto. La función Loguea puede ser extendida con muchos diferentes tipos de modems sin requerir modificación.

```
// Loguea es cerrado para modificación a través de poliformismo estático
template <typename MODEM>
void Loguea(MODEM& m, string& pno, string& usuario, string& password)
{
  m.Marca(pno);
}
```

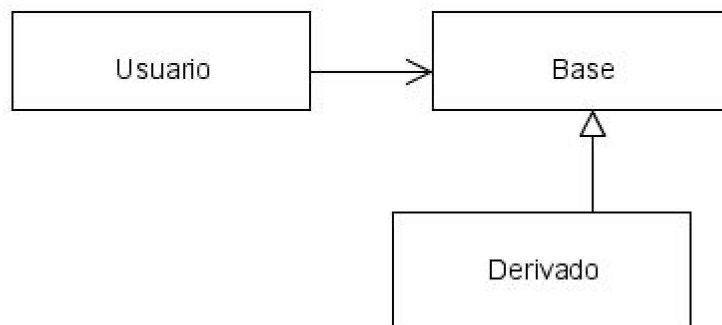
3.1.3 Metas arquitectónicas del PAC

Al utilizar estas técnicas para estar conformes al PAC, se pueden hacer módulos que son extensibles, sin ser cambiados. Esto significa que, con una pequeña previsión, se pueden añadir nuevas características al código existente sin cambiarlo y solo agregando nuevo código. Aún si el PAC no puede ser completamente conseguido, aún parcialmente puede lograr dramáticas mejoras en la estructura de la aplicación. Es siempre mejor si los cambios no se propagan en el código que actualmente existe. Si no se tiene que cambiar el código que funciona, probablemente no se tendrá que quebrar.

3.2 El principio de sustitución de Liskov (PSL)

Este principio fue acuñado por Barbara Liskov en su trabajo acerca de la abstracción de datos y teoría de tipos. También se deriva del concepto de diseño por contrato (Design by Contract, DBC) de Bertrand Meyer. El concepto, como se representa en la figura 2 establece que las clases derivadas deberían ser sustituibles por sus clases base. Esto es que un usuario de una clase base debería de continuar funcionando apropiadamente si un derivado de esa clase base le es pasada.

Figura 2. Esquema de principio de sustitución de Liskov.



En otras palabras, si alguna función de usuario toma un argumento sobre el tipo Base como se muestra en el siguiente código, debería ser legal pasar una instancia del Derivado a esa función.

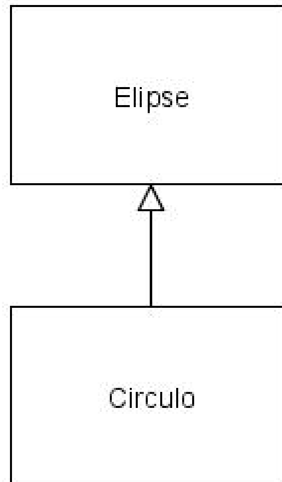
```
// Ejemplo de Usuario, Base y Derivado
void Usuario(Base& b);
Derivado d;
Usuario(d);
```

Esto podría parecer obvio, pero hay sutilezas que necesitan ser consideradas. El ejemplo canónico es el del circulo/elipse.

3.2.1 El dilema del círculo/elipse:

Como sabemos de matemática, un círculo es solamente una forma degenerada de una elipse. Todos los círculos son elipses con centros coincidentes. Esta es una relación que tienta a modelar círculos y elipses utilizando herencia como se muestra en la figura 3.

Figura 3. Dilema del círculo y el elipse



Mientras esto satisface el modelo conceptual, existen ciertas dificultades. Una mirada más cercana a la declaración de Elipse en la figura 4 empieza a exponerlo. Nótese que la elipse tiene tres elementos de datos. Los primeros dos son los centros, y el último es la longitud del eje mayor. Si el círculo hereda del elipse, entonces también heredará estas variables de datos. Esto es desafortunado, ya que el círculo realmente solo necesita dos elementos de datos, un punto de centro y un radio.

Figura 4. Declaración del elipse

Elipse
- suCentroA: Punto - suCentroB: Punto - suEjeMayor: double
+ Circunferencia(): double + Area(): double + tomarCentroA(): Punto + tomarCentroB(): Punto + tomarEjeMayor(): double + tomarEjeMenor(): double + ponerCentros(a: Punto, b:Punto) + ponerEjeMayor(double)

Aún, si ignoramos el ligero exceso en el espacio, se puede hacer que un círculo se comporte apropiadamente al obviar el método ponerCentros() para asegurarse que ambos centros son mantenidos con el mismo valor. Véase en el siguiente código, de esta manera, el foco actuará como el centro del círculo y el eje mayor será su diámetro.

```
// manteniendo la coincidencia de centros de círculo
void Circulo::ponerCentros(const Punto& a, const Punto& b)
{
  suCentroA = a;
  suCentroB = a;
}
```

Ciertamente el modelo que se ha hecho es consistente en sí mismo. Una instancia de Circulo obedecerá todas las reglas de un circulo. No hay nada que se pueda hacer para violar esas reglas. De la misma manera para Elipse. Las dos clases forman un modelo muy bien consistente, aún si Circulo tiene

demasiados elementos de datos. Sin embargo, Circulo y Elipse no viven solos en un universo por sí mismos. Ellos cohabitan ese universo con muchas otras entidades, y proveen sus interfaces públicas a esas entidades. Esas interfaces implican un contrato. El contrato puede no ser explícitamente establecido, pero está allí sin embargo. Por ejemplo, usuarios de Elipse tienen el derecho de esperar el que sea correcto el siguiente fragmento de código:

```
void f(Elipse& e)
{
  Punto a(-1,0);
  Punto b(1,0);
  e.ponerCentros(a,b);
  e.ponerEjeMayor(3);
  assert(e.tomarCentroA() == a);
  assert(e.tomarCentroB() == b);
  assert(e.tomarEjeMayor() == 3);
}
```

En este caso se espera que la función trabaje con un Elipse. De esta forma, se espera que ponga los centros, un eje mayor y entonces verifique que han sido apropiadamente configurados. Si se pasa una instancia de Elipse a esta función, será bastante afortunado. Sin embargo, si se pasa una instancia de Circulo dentro de la función, fallará. Si se fuera a hacer el contrato de Elipse explícitamente, se vería una post condición en ponerCentros() que garantizaría que los valores de entrada son copiados a las variables miembros, y que la variable del eje mayor fue dejada sin cambio. Claramente Circulo viola esta garantía porque ignora la segunda variable de entrada de ponerCentros().

3.2.2 Diseño por contrato

Reformulando el PSL, se puede decir que para que sea sustituible, el contrato de la clase base debe ser honrado por la clase derivada. Ya que Circulo no honra el contrato implicado por Elipse no es sustituible y viola el PSL. Para establecer el contrato de un método, se declara lo que debe ser verdadero antes que el método sea llamado. Esta es llamada la precondición. Si la precondición falla, el resultado del método es indefinido, y el método no debe ser llamado. También se declara que las garantías del método serán ciertas una vez que ha sido completado. Esta es llamada la post condición. Un método que falla en su post condición no debería retornar resultados. Reformulando el PSL una vez mas, esta vez en términos de los contratos, una clase derivada es sustituible por su clase base si:

Sus precondiciones no son más fuertes que el método de clase base.
Sus post condiciones no son más débiles que el método de clase base.

O en otras palabras, los métodos derivados no deberían esperar más y no proveer menos.

3.3 El principio de inversión de dependencia (PID)

Si el principio abierto/cerrado establece el objetivo de la arquitectura orientada a objetos, el DIP establece el mecanismo primario. La inversión de dependencia es la estrategia de depender de interfaces o funciones abstractas y clases, en lugar de hacerlo en funciones concretas y clases. Este principio es la fuerza habilitadora detrás del diseño de componentes, COM, EJB, CORBA etc.

El diseño procedural exhibe una clase particular de estructura de dependencia. Como se muestra en la figura 5, esta estructura comienza arriba y baja hacia los detalles. Los módulos de alto nivel dependen los módulos de nivel más bajo, los cuales dependen de módulos de nivel aún más bajo, etc.. Una pequeña reflexión expondrá esta dependencia como intrínsecamente débil. Los módulos de alto nivel tratan con las políticas de alto nivel de la aplicación. A estas políticas generalmente les importan poco los detalles que las implementan. ¿Por qué entonces, deben estos módulos de alto nivel depender directamente de esos módulos de implementación?. Una arquitectura orientada a objetos muestra una muy diferente estructura de dependencia, una en la cual la mayoría de dependencia apunta hacia abstracciones. Es más, los módulos que contienen implementación detallada no son más de los que dependen, en lugar de eso, ellos dependen de abstracciones. De esta forma, la dependencia entre ellos ha sido invertida. Véase la figura 6.

Figura 5. Estructura de dependencia de una arquitectura procedural

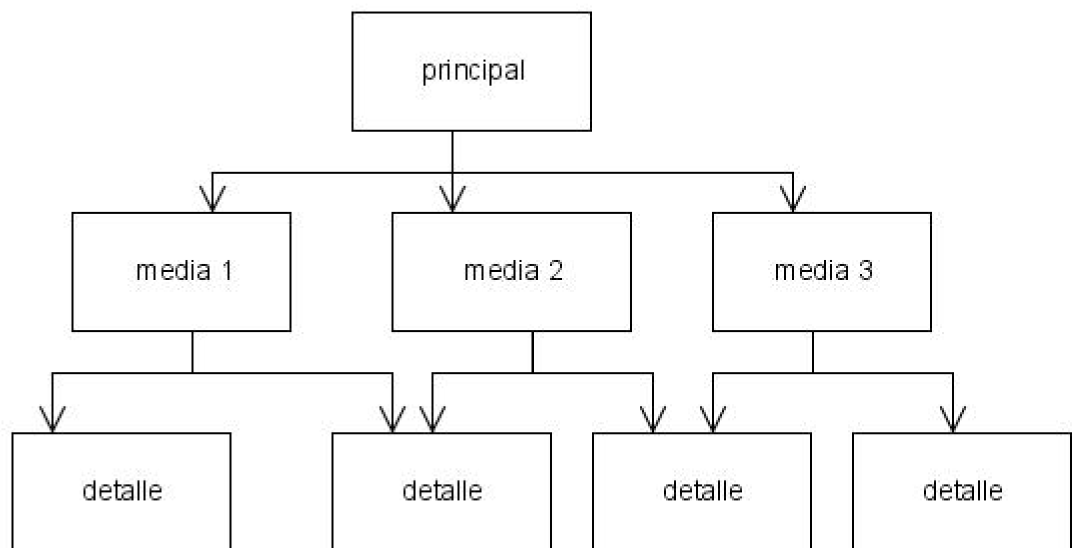
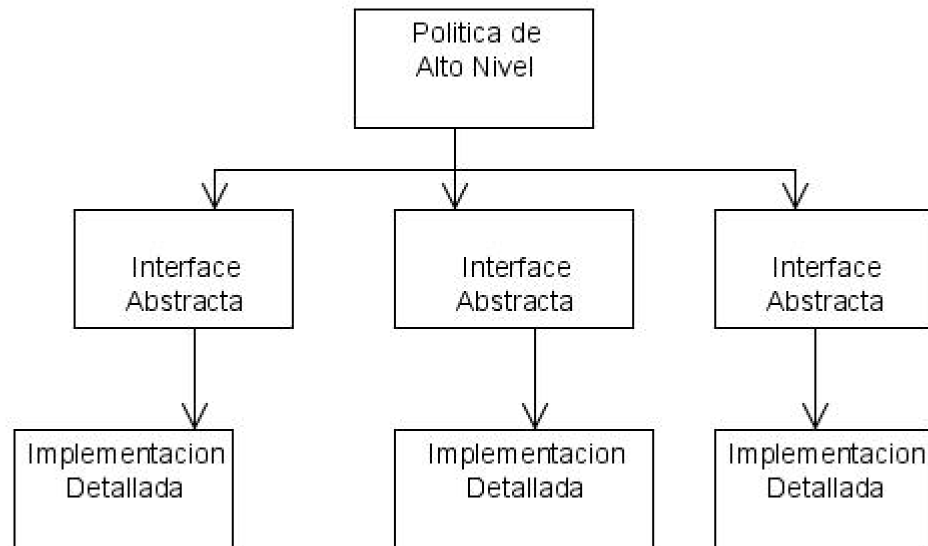


Figura 6. Estructura de dependencia de una arquitectura orientada a objetos



Claramente tal restricción es draconiana y hay circunstancias mitigantes, pero tanto como sea posible el principio debe ser seguido. La razón es simple, las cosas concretas cambian mucho y las cosas abstractas mucho menos frecuentemente. Es más, las abstracciones son “puntos de bisagra”, representan los lugares donde el diseño puede ser torcido o extendido, sin modificarse (principio abierto/cerrado).

Objetos como COM refuerzan este principio, al menos entre componentes. La única parte visible de un componente COM es su interfase abstracta. De este modo, en COM, hay poco escape del PID.

3.3.1 Mitigando fuerzas

Una motivación detrás del PID es prevenir de depender de módulos volátiles. El PID asume que cualquier cosa concreta es volátil. Mientras esto es frecuentemente así, especialmente en desarrollo temprano, hay excepciones.

Por ejemplo, la librería estándar string.h en C es muy concreta, pero no es volátil para nada. Supeditada en un entorno de string ANSI, no es perjudicial. De la misma manera, se han probado módulos que son concretos, pero volátiles, y al estar supeditados a ellos no es tan malo. Ya que estos probablemente no cambiarán, no es probable que inyecten volatilidad al diseño. Pero hay que tener cuidado, sin embargo que una dependencia de string.h se podría tornar muy fea cuando los requerimientos del proyecto fuercen a cambiar a caracteres UNICODE. La no volatilidad no es un reemplazo para la propiedad de sustitución en una interfase abstracta.

3.3.2 Creación del objeto

Uno de los lugares más comunes donde el diseño depende de clases concretas en cuando esos diseños hacen instancias. Por definición, no se pueden hacer instancias de clases abstractas. De esta manera, para hacer una instancia se debe depender de una clase abstracta. La creación de instancias puede ocurrir a través de la arquitectura del diseño, de esta forma, puede parecer que no hay escape y que la arquitectura completa estará llena con dependencias de clases concretas. Sin embargo, existe una solución elegante a este problema llamado AbstractFactory, el cual es un patrón de diseño.

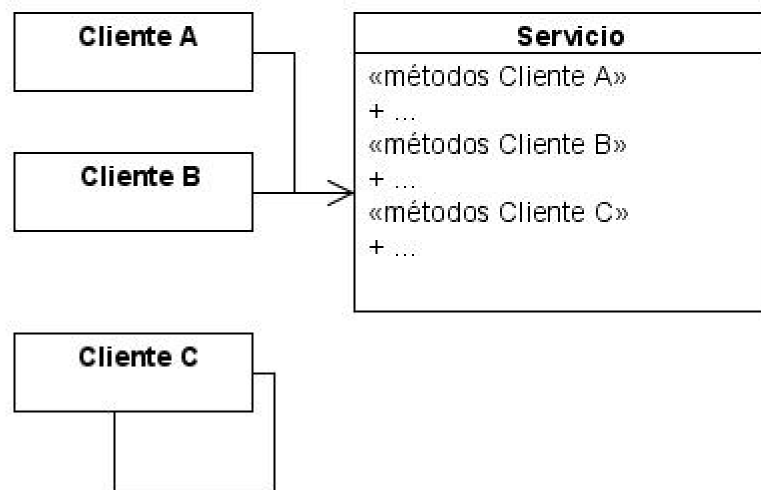
3.4 El principio de separación de la interfaz (PSI)

El PSI es otro de las tecnologías habilitadoras que soportan substracciones de componentes tales como COM. Sin esto, los componentes y clases serían mucho menos útiles y portables. La esencia del principio es bastante simple. Si se tiene una clase que tiene varios clientes, en lugar de cargar la clase con todos los métodos que el cliente necesita, se hacen

interfaces específicas para cada cliente y se multiplica la herencia de ellos en la clase.

La figura 7 muestra una clase con muchos clientes, y una larga interfaz para servir a todos ellos. Nótese que cualquier cambio que es hecho a uno de los métodos que el clienteA llama, ClienteB y ClienteC pueden ser afectados. Puede ser necesario recompilar y volver a desplegarlos, esto es desafortunado.

Figura 7. Servicio grande con interfaces integradas

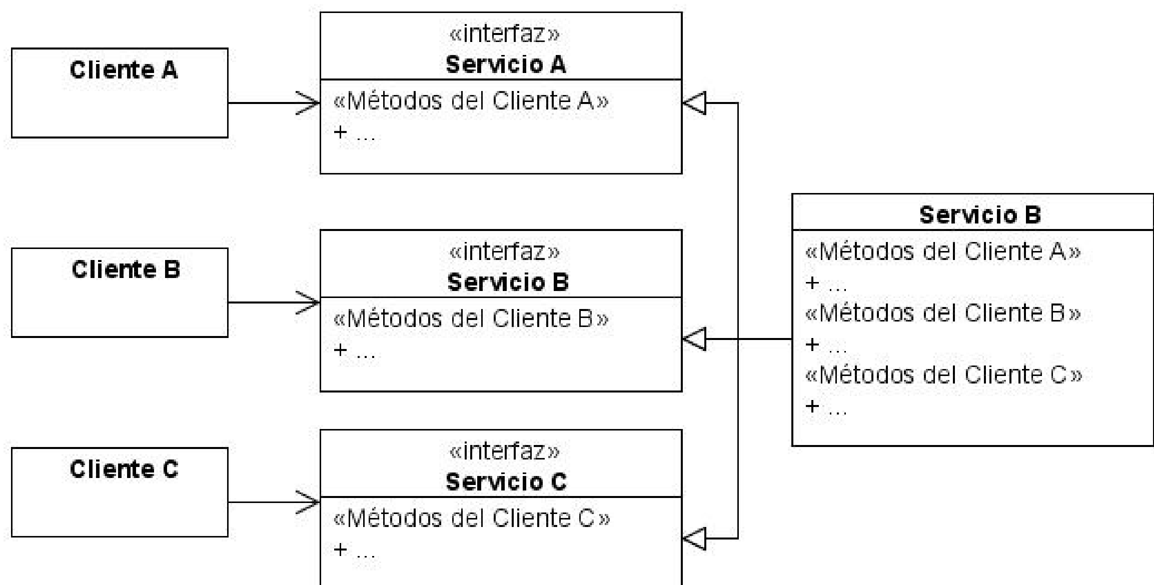


Una mejor técnica es mostrada en la figura 8. Los métodos necesitados por cada cliente son puestos en interfaces especiales que son específicas a esos clientes. Esas interfaces son múltiplemente heredadas por la clase Servicio, e implementadas allí. Si la interfaz para ClienteA necesita cambiar, ClienteB y ClienteC permanecerán sin ser afectados. No tendrán que recompilarse ni volverse a desplegar.

3.4.1 ¿Qué significa específico del cliente?|

El PSI no recomienda que cada clase que use un servicio tenga su propia interfaz de clase especial que el servicio debe heredar. Si ese fuera el caso, el servicio debería depender de todos y cada uno de los clientes de una forma bizarra y nada saludable. En cambio, los clientes deberían ser categorizados por su tipo, y las interfaces para tipo de cliente deberían ser hechas. Si dos o más diferentes tipos de clientes necesitan del mismo método, este método debería ser añadido a ambas de sus interfaces. Esto no es perjudicial ni confuso para el cliente.

Figura 8. Interfaces separadas



3.4.2 Cambiando Interfaces

Cuando las aplicaciones orientadas a objetos son mantenidas, las interfaces a clase existen y componentes a menudo cambian. Hay veces cuando estos cambios tienen un gran impacto y fuerzan la recompilación y

distribución de una gran parte de la aplicación. Este impacto puede ser mitigado al añadir nuevas interfaces a objetos existentes en lugar de cambiar la interfaz existente. Los clientes de la vieja interfaz que desean acceder métodos de la nueva, pueden consultar el objeto para esa interfaz como se muestra en el siguiente código:

```
void Cliente(Servicio* s)
{
  if (NuevoServicio* ns = dynamic_cast<NuevoServicio*>(s))
  {
  // use la interfaz del Nuevo servicio
  }
}
```

Como con todos los principios, se debe tener cuidado para no hacer trabajo innecesario. Quien espera por una clase con cientos de diferentes interfaces, algunas separadas por cliente y otras separadas por versión, de hecho debería estar aterrado.

3.5 El principio de equivalencia reutilización/publicación (PERP)

Un elemento reutilizable, ya sea un componente, una clase o un conjunto de clases, no puede ser reutilizado a menos que sea administrado por un sistema de publicación de alguna clase. Los usuarios no querrán utilizar el elemento si son forzados a actualizar cada vez que autor lo cambia. De esta forma, aunque el autor haya publicado una nueva versión de su elemento reutilizable, debe estar anuente a dar soporte y mantener versiones anteriores mientras sus clientes van por el lento camino de estar listo para actualizar. De esta forma, los clientes rehusarán a reutilizar un elemento a menos que el autor

prometa mantener un rastro de número de versiones, y mantener antiguas versiones por un tiempo. Por consiguiente, un criterio para agrupar clases dentro de paquetes es reutilización. Ya que los paquetes son la unidad de la publicación, ellos son también la unidad de reutilización. Por lo tanto, los arquitectos harían bien en agrupar clases reutilizables dentro de paquetes.

3.6 El principio de cierre común (PCC)

Un proyecto de desarrollo grande es subdivido dentro de una gran red de paquetes interrelacionados. El trabajo para administrar, probar y publicar esos paquetes no es trivial. Mientras más paquetes cambien en cualquier publicación dada, mayor será el trabajo para reconstruir, probar y distribuir la publicación. Por tanto sería deseable minimizar el número de paquetes que son cambiados en cualquier ciclo de publicación dado del producto. Para lograr esto, se agrupan juntas clases que se cree que trabajarán juntas. Esto requiere cierto grado de previsión ya que se debe anticipar las clases de cambios que habrá probablemente. Aún así, cuando se agrupan clases que cambian juntas dentro de los mismos paquetes, entonces el impacto del paquete de publicación a publicación será minimizado.

3.7 El principio de reutilización común (PRC)

Una dependencia de un paquete es una dependencia de cualquier cosa dentro del paquete. Cuando un paquete cambia, y su número de publicación es afectado, todos los clientes de ese paquete deben verificar que trabajan con el nuevo paquete, aún si nada de lo que usaban dentro del paquete cambió.

Frecuentemente se experimenta esto cuando un vendedor de sistema operativo publica uno nuevo. Se tendrá que actualizar tarde o temprano, debido

a que el vendedor no dará soporte a la versión anterior por siempre. Así que aunque nada de interés para nosotros cambió en la nueva publicación, se debe hacer el esfuerzo de actualizar y revalidar. Lo mismo ocurre con paquetes si las clases que no son usadas juntas son agrupadas. Los cambios a una clase que no interesa todavía forzarán una nueva publicación del paquete, y aún causarán que se tenga que ir al esfuerzo de actualizar y revalidar.

3.7.1 Tensión entre los principios de cohesión de paquetes

Esos tres principios son mutuamente exclusivos. No pueden ser satisfechos simultáneamente. Esto es debido a que cada principio beneficia a diferente grupo de gente. El Principio de equivalencia reutilización / publicación y el principio de reutilización común hacen la vida fácil a los reutilizadores, en tanto que el principio de cierre común hace la vida más fácil a los mantenedores. El PCC se esfuerza para hacer paquetes tan grandes como sea posible. Después de todo, si todas las clases viven en un solo paquete, entonces solo un paquete cambiará siempre. El PRC trata de hacer paquetes muy pequeños.

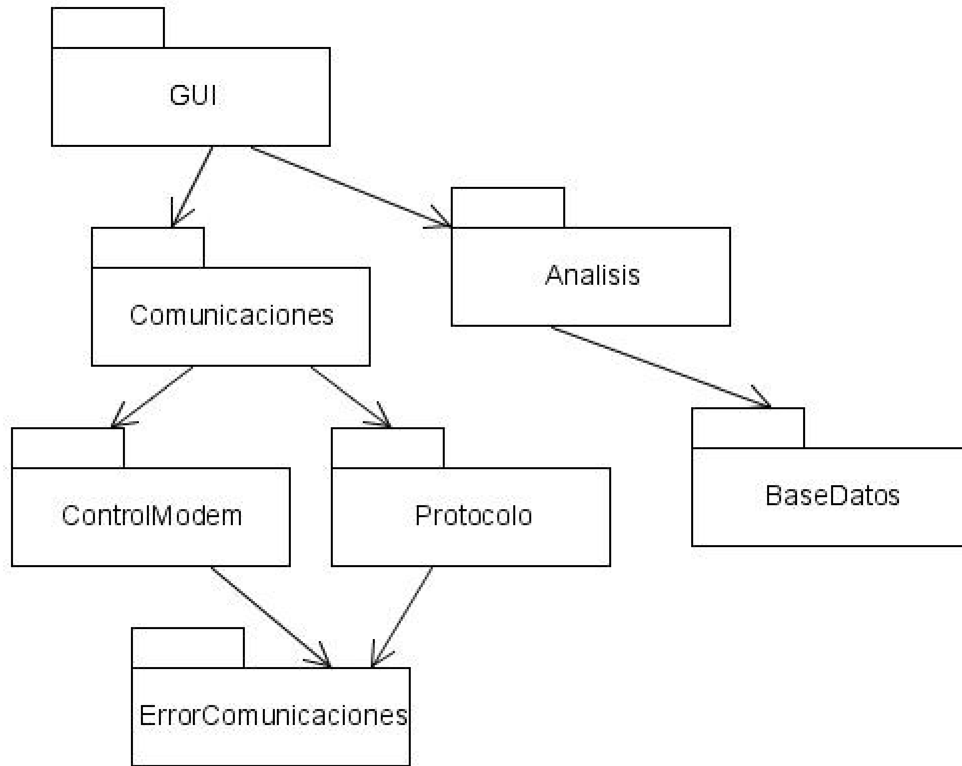
Afortunadamente los paquetes no están escritos en piedra. De hecho, es la naturaleza de los paquetes cambiar y modificarse durante el curso del desarrollo. Temprano en un proyecto, los arquitectos pueden preparar la estructura de paquetes de tal manera que el PCC domine y el desarrollo y mantenimiento sea ayudado. Más tarde, en tanto la arquitectura se estabiliza, los arquitectos pueden refactorizar la estructura de paquetes para maximizar el principio de equivalencia reutilización / publicación y el principio de cierre común para reutilizadores externos.

3.8 El principio de dependencia acíclica (PDA)

Ya que los paquetes son los gránulos de una publicación, también tienden a enfocarse en el personal. Los ingenieros típicamente trabajan en un solo paquete en lugar de trabajar en docenas. Esta tendencia es amplificada por los principios de cohesión de paquetes, ya que tienen a agrupar juntos esas clases que están relacionadas. De esta forma, los ingenieros encontrarán que sus cambios son dirigidos a unos pocos paquetes. Una vez que estos cambios son hechos, se pueden publicar esos cambios al resto del proyecto. Antes que se pueda hacer esta publicación, sin embargo, se debe probar que el paquete funciona. Para hacer eso, se debe compilar y construir con todos los paquetes de los que depende. Con un poco de suerte este número es pequeño.

Considérese la figura 9. Se puede reconocer que hay unas fallas en la arquitectura. El PID parece haber sido abandonado y junto con él el PAC. El GUI depende directamente del paquete de comunicaciones y aparentemente es responsable de transportar datos al paquete de análisis

Figura 9. Red de paquetes acíclicos



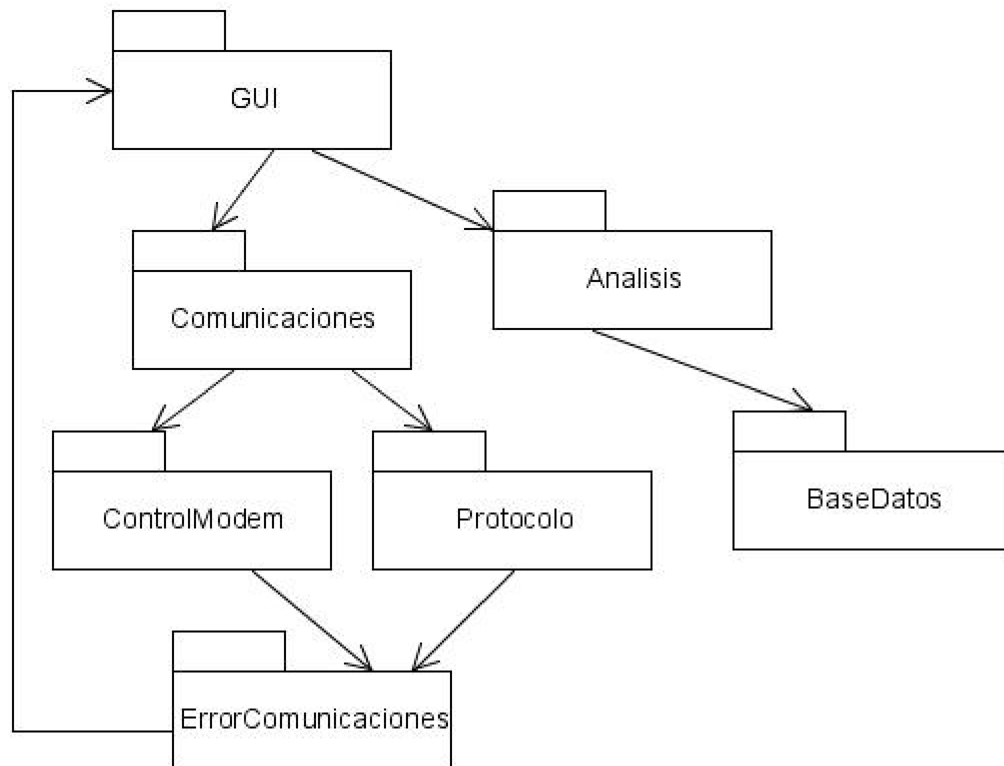
Aún así, se utilizará esta estructura para algunos ejemplos. Considérese qué se requeriría para publicar el paquete Protocolo. Los ingenieros tendrían que construirlo con la última publicación del paquete ErrorComunicaciones, y correr sus pruebas. Protocolo no tiene otras dependencias, así que ningún otro paquete es necesario. Esto es bueno, se puede probar y publicar con un mínimo de trabajo.

3.8.1 Un ciclo se introduce

Pero ahora supongamos que hay un ingeniero trabajando en el paquete ErrorComunicaciones. Ha decidido que necesita desplegar un mensaje en la pantalla. Ya que la pantalla es controlado por el GUI, envía un mensaje a uno

de los objetos GUI para tener el mensaje en pantalla. Esto significa que ha hecho a ErrorComunicaciones dependiente de GUI, véase la figura 10.

Figura 10. Un ciclo ha sido agregado

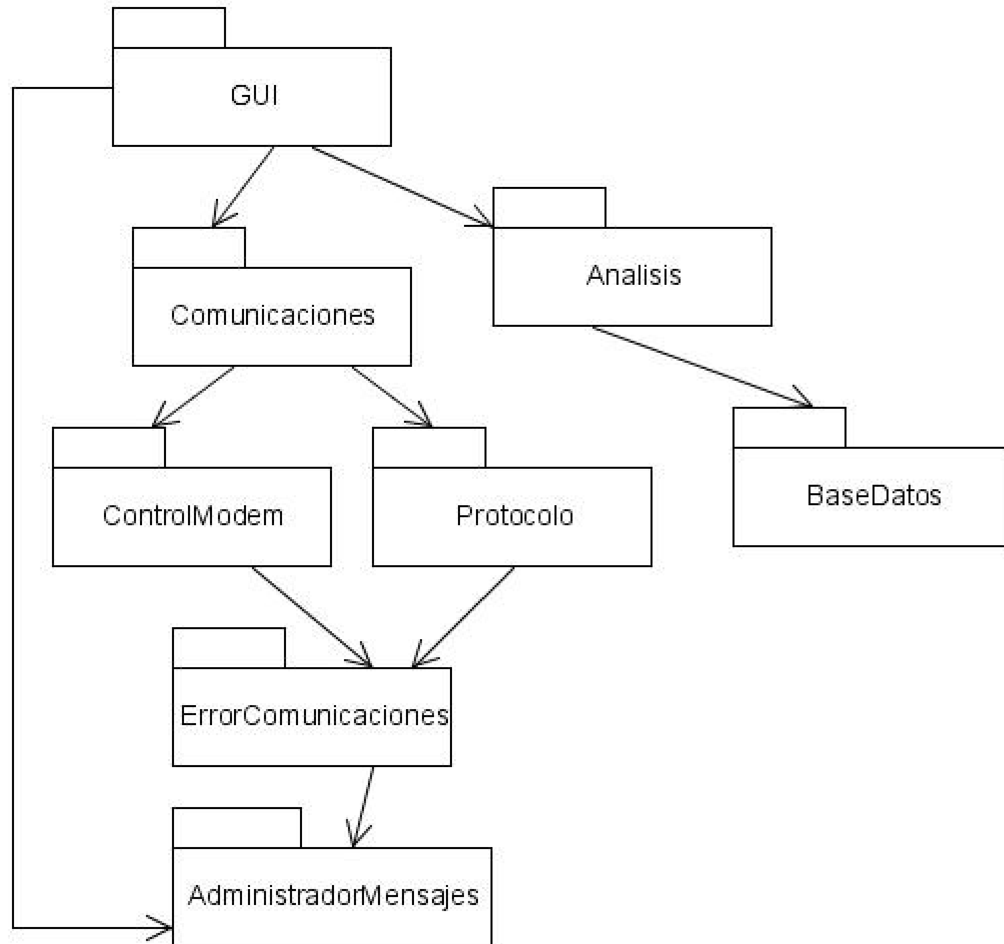


Ahora qué ocurre cuando quienes están trabajando en Protocolo desean publicar su paquete. Tienen que construir su juego de pruebas con ErrorComunicaciones, GUI, Comunicaciones, ControlModem, Análisis y BaseDatos, esto es claramente desastroso. La carga de trabajo de los ingenieros ha sido incrementada de forma aberrante, debido a una sola pequeña dependencia que se fue de control. Esto significa que alguien necesita estar vigilando la estructura de dependencia del paquete con regularidad y quebrar ciclos donde aparezcan. De otra manera las dependencias transitivas entre módulos causarán que cada módulo dependa de los otros módulos.

3.8.2 Quebrando un ciclo

Los ciclos pueden ser quebrados de dos maneras. La primera involucra hacer un nuevo paquete, y la segunda hace uso del principio de inversión de dependencia y el principio de separación de la interfaz. La figura 11 muestra cómo quebrar el ciclo al agregar un nuevo paquete. Las clases que ErrorComunicaciones necesitaba son sacadas de GUI y puestas en un nuevo paquete llamado AdministradorMensajes. Ambos GUI y ErrorComunicaciones están hechos para depender de este nuevo paquete.

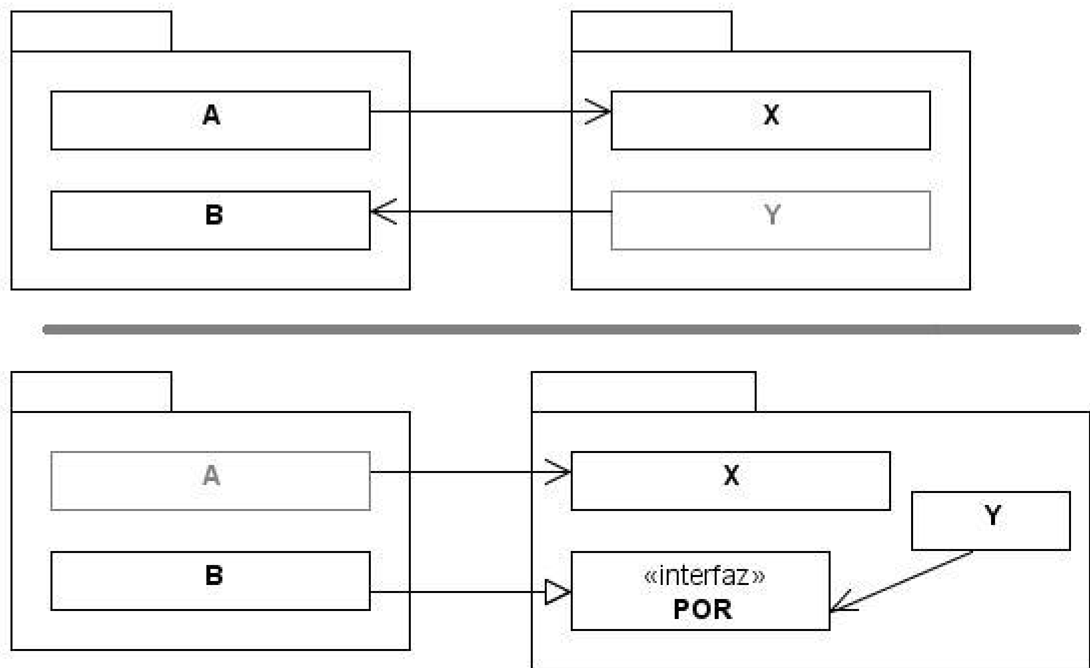
Figura 11. Ciclo quebrado



Este es un ejemplo de cómo la estructura de paquete tiende a cambiar durante el desarrollo. Nuevos paquetes vienen, y las clases se mueven de antiguos paquetes a nuevos paquetes para quebrar ciclos. La figura 12 muestra una imagen antes y después de la otra técnica para quebrar ciclos. Aquí se ven dos paquetes que están unidos por un ciclo. La clase A depende de la clase X y la clase Y depende de la clase B. Se quiebra el ciclo al invertir la dependencia entre Y y B. Esto es hecho al añadir una nueva interfaz, BY a B. Esta interfaz tiene todos los métodos que necesita Y. Y usa esta interfaz y B la implementa.

Nótese la colocación de POR. Está situado en el paquete con la clase que lo usa. Este es un patrón que se verá repetido a lo largo de los casos de estudio que tratan con paquetes. Las interfaces son muy a menudo incluidas dentro del paquete que las usa, en lugar del paquete que las implementa.

Figura 12. Otra técnica



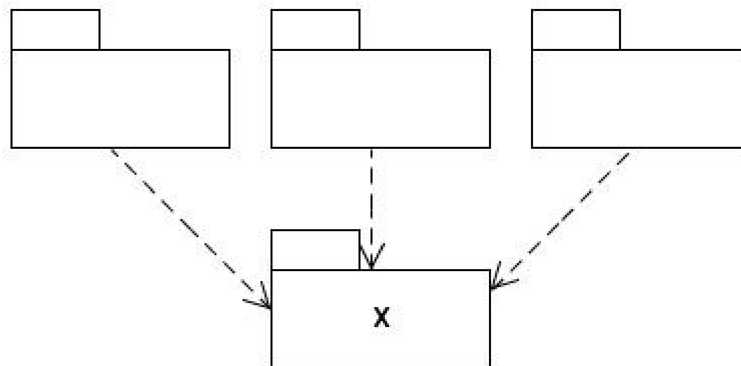
3.9 El principio de las dependencias estables (PDE)

Aunque este parece ser un principio obvio, hay algo que se puede decir acerca de esto. La estabilidad no siempre es bien entendida.

3.9.1 Estabilidad

¿Qué se entiende por estabilidad? Mantener un centavo sobre su lado. ¿Estará estable en esa posición? Posiblemente muchos dirán que no. Sin embargo, a menos que sea cambiado, permanecerá en esa posición por un muy largo tiempo. De esta forma, la estabilidad no tiene nada que ver directamente con la frecuencia del cambio. El centavo no está cambiando, pero es difícil pensar que está estable. La estabilidad está relacionada con la cantidad de trabajo requerida para hacer un cambio. El centavo no es estable porque requiere muy poco trabajo para volcarlo. Por otro lado, una mesa es muy estable debido a que toma un considerable esfuerzo darle vuelta. ¿Cómo se relaciona esto con el software? Hay muchos factores que hacen a un paquete de software difícil de cambiar. Su tamaño, complejidad, claridad, etc.. Se van a ignorar todos esos factores y se va a hacer enfoque en algo diferente. Un camino seguro para hacer que un paquete de software sea difícil de cambiar, es hacer que muchos otros paquetes de software dependan de él. Un paquete con muchas dependencias entrantes es muy estable debido a que requiere de gran trabajo para conciliar cualquier cambio con todos los paquetes dependientes.

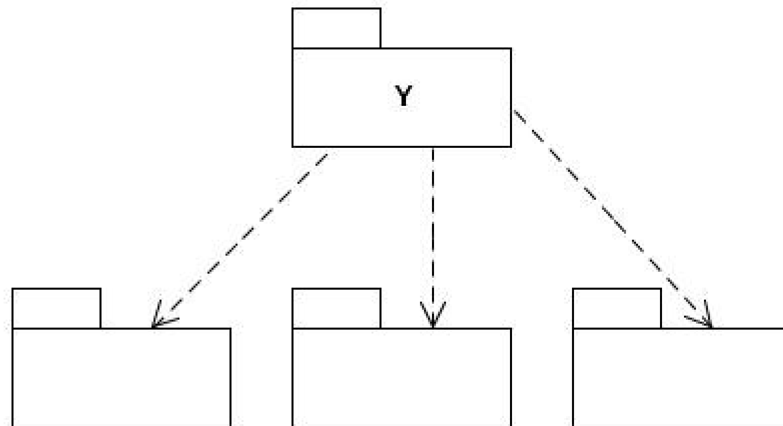
Figura 13. X es un paquete estable



La figura 13 muestra a X, un paquete estable. Este paquete tiene otros tres dependientes de él, y por lo tanto tiene tres buenas razones para no cambiar. Se puede decir que es responsable de esos tres paquetes. Por otro lado, X no depende de nada, así que no tiene influencia externa para cambiar. Se dice que es independiente.

La figura 14 muestra un paquete muy inestable. Y no tiene otros paquetes que dependen de él, se puede decir que no es responsable. Y también tiene tres paquetes de los cuales depende, así que los cambios pueden venir de esas tres fuentes externas. Se dice que Y es dependiente.

Figura 14. Y es inestable



3.9.2 Métricas de estabilidad

Se puede calcular la estabilidad de un paquete utilizando un trío de simples métricas.

Ca Acoplamiento aferente. La cantidad de clases fuera del paquete que dependen de clases dentro del paquete, es decir dependencias entrantes.

Ce Acoplamiento eferente. La cantidad de clases fuera del paquete de las cuales las clases dentro del paquete dependen, es decir dependencias salientes.

I Inestabilidad. $I = \frac{Ce}{Ca + Ce}$ esta es una métrica que tiene un rango [0,1].

Si no hay dependencias salientes, entonces I será 0 y el paquete es estable. Si no hay dependencias entrantes entonces I será 1 y el paquete es inestable. Ahora se puede expresar este principio de otra manera: "Depender de paquetes cuya métrica I es menor que la propia".

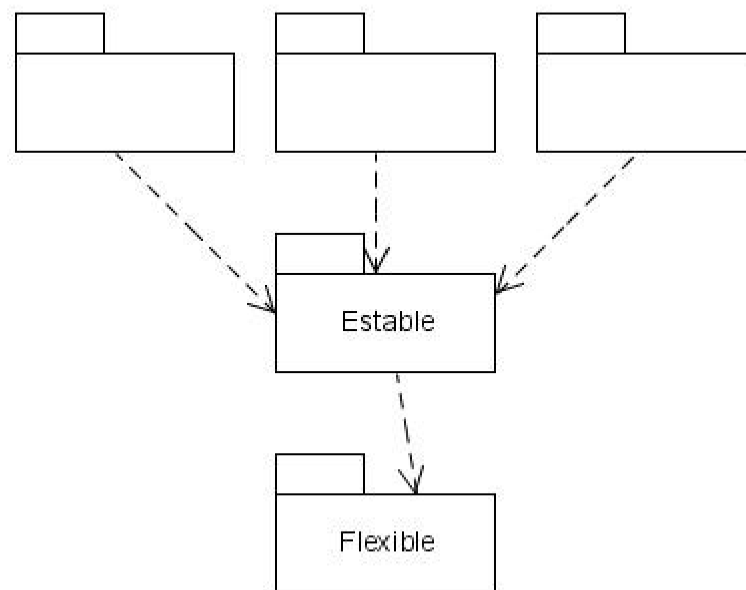
3.9.3 Fundamento

¿Debería ser todo el software estable? Uno de los atributos más importantes del software bien diseñado es que sea fácil de cambiar. El software que es flexible en la presencia de requerimientos cambiantes es tenido por bueno. Aunque tal software es inestable por definición. De hecho, se desearía que porciones del software fueran inestables. Se desea que ciertos módulos sean fáciles de cambiar para que cuando los requerimientos cambien de rumbo, el diseño puede responder con facilidad.

La figura 15 muestra cómo el PDE puede ser violado. Flexible es un paquete que se pretende que sea fácil de cambiar. Se desea que Flexible sea inestable. Sin embargo, algunos ingenieros, trabajando en el paquete llamado Estable, pusieron una dependencia en Flexible. Esto viola el principio de

dependencias estables, ya que la métrica I para Estable es mucho menor que la métrica I para Flexible. Como resultado, Flexible ya no será más fácil de cambiar. Un cambio a Flexible forzará a tratar con Estable y todas sus dependencias.

Figura 15. Violación del principio de dependencias estables



3.10 El principio de las abstracciones estables (PAS)

Al hacer una aplicación se puede prever la estructura de paquetes como un conjunto de paquetes interconectados con unos inestables arriba, y paquetes estables abajo. En esta visión, todas las dependencias apuntan hacia abajo. Aquellos paquetes que están arriba son inestables y flexibles. Pero aquellos abajo son muy difíciles de cambiar. Esto lleva a un dilema: ¿se desean paquetes en el sistema que son difíciles de cambiar? Claramente, mientras más paquetes hayan que son difíciles de cambiar, menos flexible será el diseño total. Sin embargo hay una laguna sobre la cual podemos avanzar. Los paquetes altamente estables debajo de la red de dependencia pueden ser muy

difíciles de cambiar, pero de acuerdo al principio abierto/cerrado no tienen que ser difíciles de extender.

Si los paquetes estables al final son altamente abstractos, pueden ser fácilmente extendidos. Esto significa que es posible componer la aplicación de paquetes inestables que son fáciles de cambiar, y paquetes estables son fáciles de extender. De esta manera el principio de las abstracciones estables es una reafirmación del principio de inversión de dependencia. Esto establece que los paquetes que son los más dependientes (estables) deberían también ser los más abstractos. Pero, ¿cómo se mide la abstracción?

3.10.1 Métricas de abstracción

Se pueden derivar otro trío de métricas para ayudar a calcular la abstracción.

Nc Número de clases en el paquete.

Na Número de clases abstractas en el paquete. Hay que recordar que una clase abstracta es una clase con al menos una interfaz pura, y no puede ser instanciada.

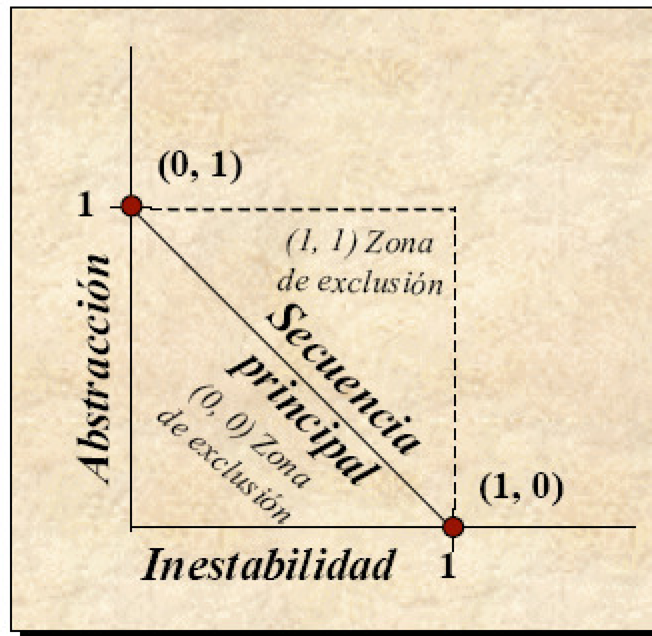
A Abstracción. $A = \frac{Na}{Nc}$.

La métrica A tiene un rango de [0,1] tal como la métrica I. El valor de A como 0 significa que el paquete no contiene clases abstractas. Un valor de 1 significa que el paquete no contiene más que clases abstractas.

3.10.2 La gráfica de I versus A

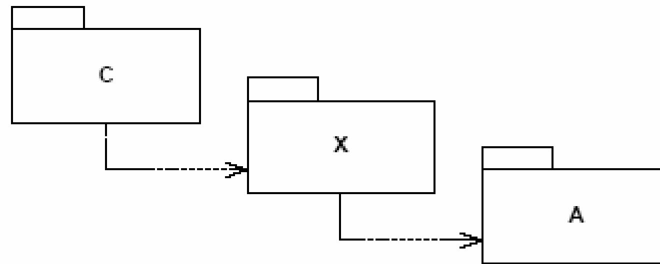
El principio de abstracciones estables puede ser ahora reafirmado en términos de las métricas de A e I: I debería incrementarse tanto que A decrece. Esto es, que los paquetes concretos deberían ser inestables mientras que los paquetes abstractos deberían ser estables. Esto se puede trazar gráficamente en la gráfica A versus I de la figura 16.

Figura 16. Gráfica de A versus I



Parece claro que los paquetes deberían aparecer en alguno de los dos lados de la figura 16. Aquellos en la parte de arriba a la izquierda son completamente abstractos y muy estables. Aquellos en la parte abajo a la derecha son completamente concretos y muy inestables. Esta es justa la manera que se desean. Sin embargo, ¿qué hay del paquete X en la figura 17? ¿dónde debería ir?

Figura 17. ¿Dónde debería ir X en la gráfica A vs I?



Se puede determinar donde se desea el Paquete X, al ver a donde no se desea que vaya. La esquina superior derecha de la la gráfica AI representa paquetes que son altamente abstractos y de los que nadie depende. Esta es la zona de inutilidad. Ciertamente no se desea que X esté en esa zona. Por el otro lado, el punto de más abajo a la izquierda de la gráfica representa paquetes que son concretos y tienen cantidades de dependencias entrantes. Este punto representa el peor caso para un paquete. Ya que los elementos allí son concretos, no pueden ser extendidos en la forma en que las entidades abstractas pueden; y ya que tienen montones de dependencias entrantes, el cambio puede ser muy doloroso. Esta es la zona de dolor, y ciertamente no se desea que el paquete esté en esa zona. Maximizando la distancia entre estas dos zonas nos da una línea llamada la secuencia principal. Sería deseable que los paquetes estuvieran situados en esta línea lo más posible. Una posición en esta línea significa que el paquete es abstracto en proporción a sus dependencias entrantes y es concreto en proporción a sus dependencias salientes. En otras palabras, las clases en tal paquete están conforme el principio de inversión de dependencia.

3.10.3 Métricas de distancia

Esto es un juego de métricas más para examinar. Dados los valores A e I de cualquier paquete, se busca saber cuán lejos el paquete está de la secuencia principal.

D Distancia. $D = \frac{|A + I - 1|}{\sqrt{2}}$ Esta tiene un rango de [0,~0.707].

D' Distancia normalizada. $D' = |A + I - 1|$. Esta métrica es mucho más conveniente que D ya que el rango es de [0,1]. Cero indica que el paquete es directamente sobre la secuencia principal. 1 indica que el paquete está tan lejos como es posible de la secuencia principal.

Estas métricas miden la arquitectura orientada a objetos. Son imperfectas y depender de ellas como el único indicador de una arquitectura fuerte sería imprudente. Sin embargo, pueden ser y han sido, usadas para ayudar a medir la estructura de dependencia de una aplicación.

4 MÉTODOS DE REUTILIZACIÓN DE SOFTWARE

El desarrollo de software ha sido dividido tradicionalmente en fases tales como análisis, diseño, implementación, pruebas, mantenimiento y diferentes ciclos de vida han sido propuestos para ayudar a manejar la dificultad de los procesos de desarrollo de software.

El desarrollo de software es un proceso consistente de una secuencia de pasos parcialmente ordenados partiendo de especificaciones más o menos formales que llevan eventualmente a un código ejecutable. Cada paso en este proceso es una transformación de una descripción de un problema en un nivel i dentro de un problema de nivel i + 1. Con este punto de vista de desarrollo de software, los métodos de reutilización pueden ser divididos en dos grupos:

- Métodos generativos: los cuales se concentran en reutilizar las transformaciones de descripciones de problemas y

- Métodos composicionales: los cuales se concentran en reutilizar las descripciones de problemas como bloques de construcción para el desarrollo de software en curso

4.1 Métodos generativos de reutilización

Los métodos generativos de reutilización automatizan una parte del desarrollo de software, es decir, algunas de las transformaciones diseñadas y ejecutadas durante el desarrollo de software. La idea de enfoques generativos es muy similar a aquella de programación automática, una especificación formal de alto nivel debería ser compilada dentro de una implementación. Sin embargo, mientras la programación automática trata de automatizar el

desarrollo de software desde la especificación a la implementación. La reutilización generativa en cambio deja el objetivo el automatizar la secuencia completa de transformaciones durante el proceso de desarrollo de software o el dominio de aplicación es estrechado.

Existen tres enfoques de reutilización que son generativos por naturaleza:

1) Sistemas basados en lenguaje.
2) Generadores de aplicación.
3) Sistemas transformacionales.

4.1.1 Sistemas basados en lenguaje

Los sistemas basados en lenguajes son también conocidos como Lenguajes Ejecutables de Especificación o lenguajes de muy alto nivel (VHLL's). Usan generalmente modelos matemáticos tales como una teoría de conjuntos, o ecuaciones restrictivas que abarcan patrones reutilizables en un lenguaje de alto nivel. El problema con los VHLL es que tienen la ventaja de ser aplicables generalmente al desarrollo de software pero la brecha semántica entre la abstracción matemática utilizada en el modelo y el desarrollo de software es aún muy grande.

SETL (SET language) es un VHLL de amplio espectro basado en la teoría de conjuntos. Los tipos de datos en el lenguaje son ya sea tipos atómicos (entero, reales o valores booleanos) o tipos de datos compuestos (conjuntos o tuplas). Las operaciones de conjuntos unión, intersección, diferencia y elevación de conjuntos, así como los iteradores de conjuntos están definidos en el lenguaje. Un mapa enlaza los elementos en el dominio a aquellos en el rango.

MODEL Es una restricción declarativa basada en VHLL resolviendo simultáneamente una colección de ecuaciones restrictivas. El lenguaje contiene construcciones para declaración de datos y ecuaciones de consistencia. Un compilador ejecuta un análisis para determinar si las ecuaciones de consistencia pueden ser satisfechas y para encontrar el orden de cómputos que asignarán valores consistentes a los objetos de datos. El compilador MODEL chequea la completitud y consistencia de las especificaciones.

4.1.2 Generadores de aplicación

Los generadores de aplicación automatizan el proceso de desarrollo completo en un dominio de aplicación muy estrecho. Un generador de aplicación es una herramienta que toma especificaciones de entrada y produce programas ejecutables como salida. Las especificaciones son típicamente abstracciones de alto nivel en un dominio de aplicación específico.

Construir un generador de aplicación es difícil, y requiere identificar los dominios apropiados, definir los límites del dominio y el modelo computacional subyacente para el dominio de aplicación, para definir partes variantes e invariantes de una familia de aplicación, para definir el método de especificación de entrada y definir productos generados por el generador de aplicación.

Para construir un generador de aplicación se requiere no sólo un íntimo conocimiento del dominio de aplicación, sino también el dominio debe ser estrecho, bien entendido, con tecnología que cambie lentamente.

4.1.3 Sistemas transformacionales

Los sistemas transformacionales no automatizan completamente el proceso de desarrollo de software. La asistencia al desarrollador de software es necesaria para seleccionar entre transformaciones aplicables. Con sistemas transformacionales, el software es desarrollado en dos fases. Primero, el comportamiento semántico de un sistema de software es descrito en un lenguaje de especificación y seguidamente, las transformaciones guiadas por el usuario son aplicadas a la especificación.

Paddle es un sistema transformacional que guarda una historia de desarrollo como una secuencia de transformaciones aplicadas. Paddle provee el programa, es decir, la estructura de la secuencia de transformaciones necesarias para implementar alguna aplicación. Las decisiones de diseño que llevan a escribir el programa no son guardadas, así que las transformaciones son difíciles de entender y modificar.

Glitter es un sistema transformacional que codifica en reglas transformacionales decisiones que un desarrollador de software ha hecho durante sus aplicaciones. Para seleccionar de una colección de transformaciones reutilizables, ha sido utilizada tecnología de sistemas expertos. Las reglas abarcan conocimiento experto acerca de cómo cumplir metas durante transformaciones de programas. Cada regla consiste de una meta, estrategias y una parte de selección racional. Glitter automatiza tanto del proceso de transformación como es posible. Si las estrategias son incompletas o la selección racional falla, el sistema pide al desarrollador de software por guía.

4.2 Métodos composicionales de reutilización

La reutilización composicional es la forma más común de reutilización de software. Está basada en reutilizar componentes de desarrollos de software anteriores como los bloques de construcción para un nuevo desarrollo de software. Los componentes guardados en librerías reutilizables pueden contener cualquier unidad que es de valor potencial para un reutilizador, tales como código fuente, subsistemas, datos de prueba, documentación de usuario, diseño, plan de desarrollo, arquitectura, instrucciones de instalación, etc..

Cuando se considera un modelo transformacional de desarrollo de software los componentes reutilizados con equivalentes a descripciones de problemas a uno de los niveles durante el proceso de desarrollo de software.

La idea principal detrás de este enfoque –reutilización de cualquier componente de software previamente desarrollado- es sencilla pero hay muchas dificultades potenciales con su aplicación. Para discutir las partes cruciales de la reutilización composicional más precisamente, los siguientes aspectos de este enfoque de reutilización de software son tomados en cuenta:

- Identificación de componentes reutilizables.
- Descripción de componentes.
- Recuperación de componentes reutilizables.
- Adaptación de los componentes a necesidades específicas.
- Integración de componentes dentro del software desarrollado actualmente.

Identificación. La identificación de partes reutilizables es el primer paso hacia la reutilización de software sistemática y efectiva. Prever las futuras

oportunidades de reutilización es difícil. Requiere experiencia en desarrollo. También, sin embargo es necesario conocimiento de dominio obtenido en la base de análisis de dominio. Muchos aspectos deben ser considerados cuando las unidades reutilizables son consideradas. Entre estas están:

- Granularidad. La granularidad (también llamada tamaño) de un componente es importante. Mientras mas grande sea el componente reutilizable, más grande será la mejora en productividad de reutilización. Pero una unidad reutilizable más grande involucra muchas funciones que pueden hacer del componente más difícil de reutilizar debido a que su funcionalidad compleja puede no encajar exactamente dentro de la que se requiere.

- Categorías. Un componente reutilizable puede ser cualquier producto intermedio del proceso de desarrollo de software, es decir, no solo código fuente sino también diseño, especificaciones, requerimientos, pruebas, documentación. Mientras más temprano sea en el proceso de desarrollo de software, más alto será el nivel de reutilización.

- Generalidad. Generalmente, los componentes independientes de aplicación que pueden ser aplicados a un amplio rango de dominios de aplicación se supone que son reutilizados más frecuentemente, pero tienden a ser más difíciles de reutilizar debido a la generalidad que ofrecen.

Está todavía muy lejos de aclarar qué es lo que debería ser un componente reutilizable y cómo identificarlo. Más que ser capaces de

reconocer componentes reutilizables desde el principio, su identificación y construcción incremental parecen ser el camino que la reutilización de software debe tomar.

Descripción: La descripción de componentes facilita el entendimiento de la funcionalidad de los componentes. Una buena descripción de componentes debería expresar qué hace un componente sin saber cómo lo hace.

Además de ser abstracto, una descripción de componente debería ser clara, no ambigua y entendible. También, debería la descripción de una amplia variedad de componentes reutilizables. En la comunidad de reutilización de software, dos enfoques a la descripción de componentes pueden ser vistos:

- Modelos de componentes. Un modelo de componente es una descripción abstracta para los componentes en un dominio dado que representa todos los atributos que un componente reutilizable debería tener. Hay un esfuerzo por encontrar un modelo aceptado en general. Hasta ahora el modelo de referencia 3C ha recibido mayor aceptación. El modelo 3C distingue distintos aspectos de un componente reutilizable, su concepto por ejemplo, una descripción abstraída de la funcionalidad que el componente de software provee, su contenido, por ejemplo, una implementación que dice cómo el componente logra la funcionalidad descrita en su concepto; y su contexto que describe la relación del componente con otros sobre los cuales depende.

  El modelo de componentes REBOOT (REuse Based on Object-Oriented Techniques, reutilización basada en técnicas de orientadas a objetos por sus siglas en inglés), está basada en una clasificación de facetas. Cuatro facetas particularmente relevantes para reutilizar han

sido propuestas: abstracción, operaciones, operandos, y dependencias. Cada faceta puede tener un número arbitrario de términos dentro de su vocabulario restringido. Un componente puede ser considerado como una combinación de los términos dentro de sus facetas.

- Lenguajes de descripción de componentes. Ya sea al nivel de implementación o al nivel de diseño, estos lenguajes tratan de capturar los atributos esenciales de los componentes.

  La meta de un lenguaje de descripción de componentes es proveer semántica de la funcionalidad de los componentes. De hecho, hay dos principales enfoques a especificaciones formales: uno algebraico y un enfoque basado en modelo. Una especificación formal algebraica describe interfaces sintácticas por medio de nombres y firmas para todas las operaciones de componentes y declara las relaciones entre las operaciones. Una especificación formal basada en modelo define precondiciones y poscondiciones para cada operación.

Recuperación: Las librerías reutilizables pueden crecer hasta una gran colección de componentes. Para que no se convierta en medio de solo escritura, deben estar bien organizados. En particular, métodos para recuperación de componentes que codificarían descripción abstracta de componentes y los emparejarían con los requerimientos del reutilizador deben ser agregados a la librería.

- **Librería y ciencias de la información.** Hay dos grupos de métodos utilizados en la librería y ciencia de la información: uno que usa un vocabulario controlado, por ejemplo, enumerado, clasificación de faceta y

valor de atributos, y el otro que no restringe vocabulario, llamado también recuperación libre de texto.

La clasificación enumerada usa descripciones cortas, usualmente descripciones metafóricas de una palabra para romper ámbitos en clases mutuamente exclusivas que forman una estructura jerárquica. Esto solo es posible en bien entendidos, dominios de aplicación estrechos con abstracciones de una palabra que abarcan una gran cantidad de conocimiento de dominio de tal manera que pueden ser universalmente entendidos. La clasificación provee un método natural de búsqueda cuando el dominio es bien analizado y existen categorías jerárquicas exclusivas. Una desventaja de la clasificación enumerada es la dificultad conectada con cambiarla.

La clasificación multifacetada organiza términos de un ámbito dentro de facetas. El desarrollo de facetas es cumplido identificando vocabulario importante en un dominio y agrupando términos dentro de facetas.

La clasificación de valor de atributos usa un conjunto de atributos y sus valores para describir un componente en la librería. Es similar a la clasificación de facetas porque usa atributos y valores como la clasificación de facetas utiliza facetas y valores.

Las diferencias son que los valores de atributos no están restringidos a tener valores predefinidos como tienen las facetas y no hay límite en el número de atributos utilizados para descripción de componentes. El enfoque de vocabulario controlado tiene una desventaja que los vocabularios del

reutilizador y el desarrollador pueden diferir. Usualmente, un tesauro de sinónimos es proveído para eliminar esta diversidad.

La *recuperación libre de texto* está basada en lenguaje natural. La representación textual del componente es utilizada como una descripción de componente. Una ventaja de la recuperación libre de texto es que no se requiere codificación y consultas en lenguaje natural son fáciles de formular debido a la cercanía al reutilizador. Pero la ambigüedad del lenguaje natural, falta de completitud e inconsistencia hacen la recuperación de componentes menos precisa.

- **Recuperación basada en conocimiento.** La recuperación basada en conocimiento usa varias clases de razonamiento la nueva consulta con una antigua o que coincide la consulta con los componentes. Estos métodos necesitan una base de conocimiento para el dominio de aplicación y para las decisiones tomadas. Estos requieren más recursos humanos especialmente para el desarrollo de la base de conocimiento pero tienen un potencial para ser más poderosas porque capturan consultas y componentes semánticos.

- **Recuperación basada en hipertexto.** La recuperación basada en hipertexto organiza información no lineal dentro de una red de nodos y enlaces. Un reutilizador puede acceder la información guardada navegando a lo largo de los enlaces. Para diseñar una red de hipertexto se requiere estudiar el dominio de aplicación cuidadosamente y encontrar un conjunto óptimo de relaciones. Puede también ser difícil añadir un nuevo componente debido a que requiere estudiar exhaustivamente los vínculos entre los nuevos componentes y los antiguos. Los hipertextos son fáciles de usar y pueden llevar al componente correcto rápidamente. Pero el utilizador puede también perderse durante la navegación o alguna información puede aún ser

inaccesible debido a la ruta que ha escogido el cual no tiene enlace a la información requerida.

- **Recuperación basada en especificación.** Los enfoques basados en especificación usan descripciones formales de los componentes como base para el ordenamiento parcial de los componentes en la librería. Estos componentes difieren mayormente en la expresividad del lenguaje de especificación y también en cuán completamente toman ventaja del lenguaje de especificación. Algunos hacen uso de signos de componentes solamente, por ejemplo, interfaces sintácticas; los otros manejan también las semánticas de especificaciones

No existe acuerdo acerca de cuál método de recuperación es el más adecuado. Un simple método de recuperación no es suficiente para encontrar todos los componentes relevantes para una consulta de búsqueda dada. Diferentes usuarios individuales prefieren y son más exitosos con diferentes métodos. Ninguno de los métodos soporta el entendimiento de los componentes más que moderadamente. Inventar métodos de recuperación que se ajusten mejor para componentes reutilizables permanece como un reto.

- **Formulación de consultas.** Es otro importante asunto en el método de recuperación. Aquí, la indisposición del recuperador de formular largas y precisas consultas debería ser tomada en cuenta. El reutilizador preferiría una formulación de consultas interactiva donde un sistema ayude a definir el query. El hecho que el reutilizador a menudo no es capaz de formular completamente su consulta al principio del proceso de recuperación es acentuado hacia fuera y la construcción incremental de consultas es propuesta. También debería ser considerada una recuperación aproximada

o relajada. Siempre que una coincidencia exacta falle, los componentes que son más cercanas a los requerimientos deberían ser recuperados

Adaptación..Una de las características básicas de un componente reutilizable es su generalidad. Este acto es llamado adaptación de componentes o especialización. La adaptación de componentes puede ser hecha usando diferentes técnicas:

- Substitución de parámetros. La parametrización permite construir componentes genéricos, por ejemplo componentes con parámetros que son adaptados a necesidades específicas poniendo valores a sus parámetros.

- Herencia. La herencia es un mecanismo que permite añadir nuevas propiedades a clases existentes (rasgos substanciales del paradigma orientado a objetos). Este es un ejemplo de programación incremental cuando las subclases son definidas al establecer cómo difieren de las existentes.

- Modificaciones. Las modificaciones de un código de componente es una reutilización de caja blanca que requiere entender todos los detalles de la implementación. También, esta técnica puede invalidar la corrección del componente original.

Integración. Para desarrollar un componente para reutilización y subsecuentemente reutilizar el componente, no solo el componente, sino también su relación con otros componentes debe ser bien entendido. La efectividad de la reutilización depende altamente de la manera en que esos componentes son combinados. Hay no solo un número de diversas entidades

de software componibles tales como funciones, clases , plantillas, módulos, procesos, pero consecuentemente también un número de estilos de componentes para combinar los sistemas de software desde estos componentes. Diferentes estilos suponen diferentes formas de empaquetamiento de componentes y diferentes clases de interacciones entre ellos.

La manera más común de describir la integración de componentes es usar lenguajes de interconexión de componentes. Los lenguajes de interconexión de módulos describen los módulos en términos de operaciones exportadas que un módulo implementa y operaciones importadas que un modelo usa. Los módulos son ensamblados dentro de un sistema interconectándolos a través de exportaciones e importaciones.

Los lenguajes declarativos son sugeridos en ser usados para describir la interconexión de componentes. Estandarizando ciertos mecanismos para interacción de componentes, los lenguajes declarativos pueden ser utilizados para describir los modos en que los componentes están conectados. La principal ventaja es que las interacciones de los componentes no necesitan ser descritas en detalle y solo relaciones entre componentes son establecidas. Además, los lenguajes declarativos pueden ser invertibles, de tal manera que las mismas relaciones pueden ser interpretadas de distintas maneras.

Hasta ahora la mayor parte del énfasis ha sido puesto y solo una pequeña parte en cómo están compuestos. Más investigación es necesaria en cómo componer software desde elementos computacionales. Una gran cantidad de trabajo reciente ha sido hecho en áreas tales como lenguajes de interfaz de módulos, marcos de trabajo específicos de dominio, y arquitecturas de software. En particular, la arquitectura de software como descripción de alto nivel de

cómo sistemas específicos están compuestos desde estos componentes has sido reconocido como un aspecto crítico del diseño de cualquier sistema de software más grande y ha ganado significativa atención recientemente.

4.2.1 Reutilización dentro de la metodología orientada a objetos

Recientemente, muchos investigadores y practicantes del área de reutilización de software dedicaron su atención al desarrollo de software orientado a objetos. Se cree que los lenguajes orientados a objetos facilitan el desarrollo de software que puede ser reutilizado. Aquí recientemente un fuerte ímpetu pudo ser observado en el desarrollo y rápida dispersión del lenguaje de programación Java.

Existen de hecho diferentes niveles de reutilización ofrecidos en el desarrollo de software orientado a objetos. Los objetos y clases son unidades reutilizables básicas. Pero ellos solos no pueden ser suficientes para una reutilización efectiva. No solo los componentes por sí mismos sino la manera en que esos componentes son combinados afectan a la reutilización de software significativamente. De hecho, las metodologías orientadas a objetos dieron origen a marcos de trabajo de aplicación, y más recientemente, patrones de diseño para construcción de aplicaciones orientadas a objetos. Los patrones de diseño comunican soluciones a problemas de diseño recurrentes en el desarrollo OO. Ambos, marcos de trabajo y patrones elevan la reutilización a un nivel más alto debido especialmente a que dan soporte a la reutilización de diseño.

Mientras es probablemente justo decir que ya sea la afirmación de que la orientación a objetos fomenta la reutilización es justificado permanece al menos por algún tiempo por ser visto, puede ser establecido por ahora que ha habido

más progreso substancial logrado en buscar conceptos y formas que permitan expresar diseño estándar de software dentro de la metodología orientada a objetos.

4.2.2 Reusabilidad de objetos

La unidad fundamental en el desarrollo orientado a objetos es el objeto. Los conceptos básicos envueltos en la noción de objeto son abstracción de datos, ocultamiento de información y encapsulación. Los objetos son definidos en términos de tipos de dato abstracto donde cada tipo define también un conjunto de métodos. Un objeto tiene su estado interno (dato) y sus operaciones (métodos). El único medio en que otros objetos interactúan con un objeto es a través del envío de mensajes (solicitudes).

Las siguientes características de un objeto fomentan su reusabilidad:

- Objetos separan la interfaz de la implementación.
- Los objetos a menudo se acercan mucho a la realidad.
- Objetos vienen en distintos tamaños y niveles de abstracción.
- Los objetos viven completamente durante el desarrollo de software.

4.2.3 Reutilización de clases

La clase es una plantilla común para objetos similares. La clase define la interfaz también como el comportamiento de un conjunto de objetos. Los objetos instanciados de una clase ejecutan métodos comunes y comparten estructura común mientras que cada objeto dentro de una clase retiene sus propios estados.

Los lenguajes de programación OO usualmente ofrecen una librería de clases predefinidas. Las clases en la librería están relacionadas a través de relación de herencia. La herencia tiene dos posibles usos:

- **Instanciación** hace un objeto específico desde una plantilla de clase al poner valores a variables definidas en la clase. Esto es el por qué los objetos algunas veces son llamados instancias de la clase. Cuando un objeto es hecho, sus datos internos son aprovisionados de acuerdo a la estructura de la clase y las operaciones de la clase están relacionadas con esos datos. Muchos objetos pueden ser hechos instanciando la misma clase.

- **Subclaseado** permite la derivación de nuevas clases al modificar las clases existentes, por ejemplo, al establecer cómo difieren de estas. El subclaseado involucra dos tipos de relaciones entre clase y subclase:

  - **Especialización:** la especialización añade nueva funcionalidad a las subclases. Esto retiene la semántica de la interfaz, por ejemplo, una subclase de la interfaz es un subtipo de su interfaz de superclase.

  - **Omisión:** la omisión puede refinar, redefinir, o aún esconder la funcionalidad de los padres. Redefinir y esconder rompe la semántica de la interfaz.

Los beneficios de la reutilización de clases incluyen menos código a desarrollar, menos código a mantener, y no tener que rediseñar los mismos elementos repetidamente. La herencia hace más fácil incluir código existente

extendiendo las clases originales y añadiendo nuevas. Pero la herencia permite también permite redefinición y ocultamiento que podría tener efectos dañinos sobre la reutilización. Tales usos diversos del mismo mecanismo de herencia causa dificultades en la reutilización de clases.

Un mecanismo útil que la herencia permite es definir clases abstractas. Una clase abstracta no tiene implementación para sus métodos, así que no puede ser instanciado. Una clase abstracta define una interfaz común para sus subclases mientras que las definiciones concretas son dejadas a la subclase en sí misma. Esto ayuda a definir familias de clases intercambiables con interfaz común.

Las clases abstractas hacen las librerías de clases orientadas a objetos más reutilizables y fáciles de entender. El mecanismo que las clases abstractas proveen es uno de los más importantes: las clases deberían ser derivadas desde superclases abstractas tan a menudo como fuera posible y todas las subclases de una clase abstracta deberían solo añadir o refinar pero no omitir operaciones de la clase padre. Entonces, todas las clases pueden responder a la interfaz idéntica, es decir todos son subtipos de la misma clase abstracta.

También las diferentes vistas de un implementador y un reutilizador que pueden causar diversidad en la jerarquía de clases deberían ser señaladas. Un implementador está mayormente involucrado con la rápida derivación de una nueva implementación de clase desde las existentes. Un reutilizador está más interesado en la herencia de interfaz que dice cuando una clase puede ser utilizada en lugar de otra.

Otra fuente de potenciales dificultades es que las librerías de clases de pueden volver vastas. Reutilizar clases desde tal jerarquía puede ser una

laboriosa tarea que consume tiempo. Es necesario mirar a los nombres de clases, a los nombres de métodos, o en el peor de los casos a la implementación de métodos para investigar oportunidades de reutilización. La herencia es una reutilización de caja blanca que requiere que todo lo interno de una clase padre sea visible a su subclase. El ambiente OO usualmente facilita la reutilización por medio de la visualización de la estructura de jerarquía de clases y dando soporte a una fácil navegación de clases.

4.2.4 Reutilización en el mundo

La Internet, una colección de redes de rápido crecimiento, y la World Wide Web, un sistema de comunicación e información de hipertexto, dieron origen a no solo a nuevas maneras de comunicación entre la gente sino también cambiaron la industria del software significativamente.

La internet y la web últimamente requieren nuevos enfoques al desarrollo de software. La habilidad de correr sobre plataformas heterogéneas y distribuidas es la necesidad para todos los futuros sistemas de software. Al tener tal plataforma independiente de software, la reutilización se incrementa en una tasa significativa. No es necesaria más reescritura específica de código. La misma pieza de software es reutilizada a lo largo de una amplia variedad de sistemas de computación, plataformas de hardware y sistemas operativos.

Java, ha sido designado para hacer uso completo de los avances en redes distribuidas. Java es un lenguaje que trae una nueva dimensión a la reutilización de software también. Sin duda, la característica clave de Java que ayuda a la reutilización de software es su genuina orientación orientada a objetos. Pero Java introduce capacidades adicionales que soportan la reutilización:

- Java es arquitectura neutral. El código de Java es capaz de correr en cualquier plataforma que tenga el Java runtime environment. Con Java, la capacidad de reutilización es aumentada al tener una simple aplicación que es inmediatamente utilizable en múltiples plataformas.

- Java es portable. La especificación de lenguaje de Java define comportamiento estándar aplicado a los tipos de datos a través de plataformas heterogéneas. La reutilización es mejorada debido a que la implementación no es más dependiente del hardware.

- Java es dinámico y robusto. Enlace dinámico de clases en tiempo de corrida y chequeo de estructuras de datos en tiempos de compilación y corrida causan que Java pueda ser reutilizado sin recompilación aún si el entorno ha cambiado.

Hay otra característica de Java que debería ser tomada en cuenta, el código de Java puede ser una parte ejecutable de un documento Web. Java applets, por ejemplo, pequeñas aplicaciones que son bajadas directamente de páginas web, traen riqueza, interactividad, y mejoran la entrega de la información a las paginas Web. A través de internet, la reutilización es fomentada a lo largo del mundo.

4.2.5 Marcos de trabajo de aplicación

Un marco de trabajo es un diseño reutilizable para soluciones de problemas en algún particular dominio de problema. Un marco de trabajo de aplicación orientado a objetos es un conjunto de clases que tomadas juntas representan un abstracción o esqueleto parametrizado de una arquitectura.

Los marcos de trabajo agrupan clases, objetos, y relaciones juntas para construir una aplicación específica. Un marco de trabajo es una arquitectura de software genérica junto con un conjunto de componentes de software genéricos que pueden ser utilizados para realizar arquitectura de software específica.

Un marco de trabajo OO de un conjunto de clases abstractas y concretas que se comunican extensivamente a través de mensajes. Un marco de trabajo ideal debería proveer todas las clases concretas desde las cuales nuevas aplicaciones deberían ser compuestas en su librería de clases. En el desarrollo de software real basado en marcos de trabajo, algunas de las clases específicas de aplicación deben ser construidas. Son típicamente derivadas al subclasear desde clases abstractas proveídas en el marco de trabajo. Aquí, las clases abstractas representan partes variables del marco de trabajo que son configuradas de acuerdo a las necesidades específicas de una aplicación .

Hay muchos diferentes marcos de trabajo OO disponibles. Entre ellos, uno de los mejor conocidos es el Model-View-Controller (MVC) construido en el sistema Smalltalk-80. MVC es un marco de trabajo de interfaz de usuario que provee una arquitectura uniforme para aplicaciones interactivas. Una vista de clase abstracta convierte algunos aspectos interesantes del modelo a una forma visible. Un modelo en si mismo entra y actualiza la aplicación.

Diseñar un marco de trabajo requiere un análisis minucioso de dominio así como experiencia de distintos proyecto. Los asuntos más críticos en diseñar un marco de trabajo es conseguir su flexibilidad, es decir, fácil adaptación a todas las aplicaciones en el dominio, y su extensibilidad, es decir, la habilidad de cubrir todas las aplicaciones en el dominio. También, es necesaria una baja sensibilidad a la evolución del dominio.

Nuevas aplicaciones pueden ser construidas más rápido cuando los marcos de trabajo son reutilizados. Cada marco de trabajo logra su funcionalidad por medio de una cooperación de componentes. Entender y administrar un simple componente puede volverse más difícil porque esto depende de sus relaciones con otros componentes también. Aparte de esto los marcos de trabajo proveen soluciones "marco" al problema que puede no necesariamente encajar en el problema.

4.2.6 Patrones de diseño

Los patrones de diseño introducen un nuevo mecanismo para expresar conocimiento de diseño que experimentó el uso de desarrolladores sobre distintas aplicaciones. Cada patrón de diseño sistemáticamente identifica, nombres y explica un recurrente problema de diseño y presenta buenas y elegantes soluciones a esto. El patrón de diseño es una microarquitectura, un pequeña agrupación de clases describiendo responsabilidades de partes elementales así como las relaciones y colaboraciones entre ellos.

Los patrones son construidos por observación y al reunir experiencia durante el desarrollo de muchas aplicaciones orientadas a objetos. Una plantilla de patrón involucra:

- Patrón: nombre, sinónimos, descripción de una sentencia, título, alias.
- Intención: meta, motivo.
- Aplicabilidad: contexto, ejemplos.
- Consecuencias: la situación final, resultados.
- Restricciones: interdependencias, fuerzas, factores que afectan.
- Resolución: estructuras, acciones.

- Implementación: fragmentos de código, preocupaciones prácticas para estar al tanto.
- Aplicaciones: usos conocidos.
- Riesgos: indicaciones de potenciales dificultades en el uso.
- Patrones relacionados: referencias a otros patrones.

La plantilla uniforme utilizada para la descripción de patrones los hace fáciles de aprender, comparar y utilizar. Cada patrón explica una solución a un problema exhaustivamente de tal manera que ninguna información esencial es perdida para los lectores. La descripción de patrones utiliza una plantilla la cual es una estructura de atributos. La plantilla contiene cuatro elementos esenciales: un nombre de patrón, un problema, una solución y consecuencias. Diagramas de clases, diagramas de objetos y diagramas de interacción son utilizados para ilustraciones de problemas. Además de notaciones gráficas, lenguaje natural describe decisiones, alternativas, ejemplos, y productos que llevan al patrón de diseño propuesto.

Los patrones dividen las soluciones en sus partes elementales las cuales pueden ser más tarde recombinadas y reutilizadas. Los patrones describen minuciosamente una sola solución atómica. Las descripciones de las combinaciones se pueden simplificar debido a que se pueden referir a los patrones ya descritos y guardados.

Los marcos de trabajo de aplicación y los patrones de diseño son a menudo confundidos. Las principales diferencias entre marcos de trabajo y patrones son:

- Los patrones de diseño son más abstractos que los marcos de trabajo. Los marcos de trabajo pueden ser plasmados en código, pero sólo ejemplos de patrones pueden ser plasmados en código.

- Los patrones de diseño son elementos arquitectónicos más pequeños que los marcos de trabajo. Un marco de trabajo típico contiene varios patrones de diseño, pero el contrario nunca es cierto.

- Los patrones de diseño son menos especializados que los marcos de trabajo. Los marcos de trabajo siempre tienen un dominio de aplicación particular. Por el contrario, los patrones de diseño pueden ser utilizados independientemente de una aplicación.

Los patrones ocurren fuera de los metodologías orientadas a objetos también. Los patrones de diseño varían en su granularidad y nivel de abstracción. Los patrones pueden abstraer un análisis, un diseño o un proceso.

Los patrones podrían ser organizados en familias de patrones relacionados para que el reutilizador pueda aprenderlos encontrarlos más fácilmente. Cuando se coleccionan patrones, la manera de categorizar patrones en una colección debería ser propuesta también.

5 EJEMPLOS DE REUTILIZACIÓN COMPOSICIONAL CON PATRONES DE DISEÑO EN PHP

Para ejecutar los ejemplos en este trabajo de graduación se utilizó el webserver PHP Wamp en versión 2.0. También se utilizó la herramienta Dreamweaver en su versión CS3 para correr los ejemplos, así como la herramienta opensource en java UMLET versión 10.3 para realizar los diagramas UML en los ejemplos.

5.1 Uso del patrón de diseño observer

Parte de la expresividad de la programación orientada a objetos es la habilidad de construir redes complejas de interconexiones entre objetos. Vinculados juntos, los objetos pueden intercambiar servicios e información. A menudo, se desea que los objetos puedan comunicarse cuando un objeto cambia. Pero por muchas razones, se puede preferir no "quemar" el código de las líneas de comunicación. Quizás se desea formar y reformar las conexiones para responder a condiciones en la aplicación o tal vez simplemente se desea refactorizar el código de comunicación para evitar interdependencias entre clases.

5.1.1 El problema

¿Cómo se puede alertar (potencialmente) a muchos objetos cuando un cierto estado del objeto cambia? ¿Hay algún esquema dinámico, alguno que permita interconexiones entrantes y salientes al ejecutarse algún script?

5.1.2 La solución

El patrón observer permite a los objetos expresar interés en el estado de otro objeto y provee un mecanismo para el “observado”, o el sujeto de contactar a todos sus “observadores”, los clientes cuando su estado cambia.

El observer es una colaboración entre una clase observable (el sujeto) y una o más clases observadoras (los clientes). La clase observable permite a los observadores registrarse con él. Entonces, siempre que el estado del objeto observable cambia, todos los observadores registrados son notificados. El patrón observador separa al sujeto del cliente, dejando a cada observador tomar su propia acción en respuesta al cambio. (El patrón observador es también conocido como publicar/subscribir, la cual es una metáfora valida para la interacción entre los objetos en el patrón.)

El patrón observador es flexible y extensible. La carga de saber qué clases desean seguir la información del estado del observable y cómo cada una de estas clases tiene la intención de usar la información, es removida de la clase observable en sí misma. Adicionalmente, un observer puede registrar o quitar el registro en cualquier momento, si es apropiado. Se puede también definir múltiple clase observer concreta, permitiendo un comportamiento variado en la aplicación.

5.1.3 Código de ejemplo

Un ejemplo clásico es el código en un dialogo observando el estado de un checkbox. Al checkbox no le importa si está siendo observando por un objeto o por cientos. Simplemente envía un mensaje cuando cambia de estado. De la misma forma, al diálogo no le importa cómo el checkbox es implementado, sólo

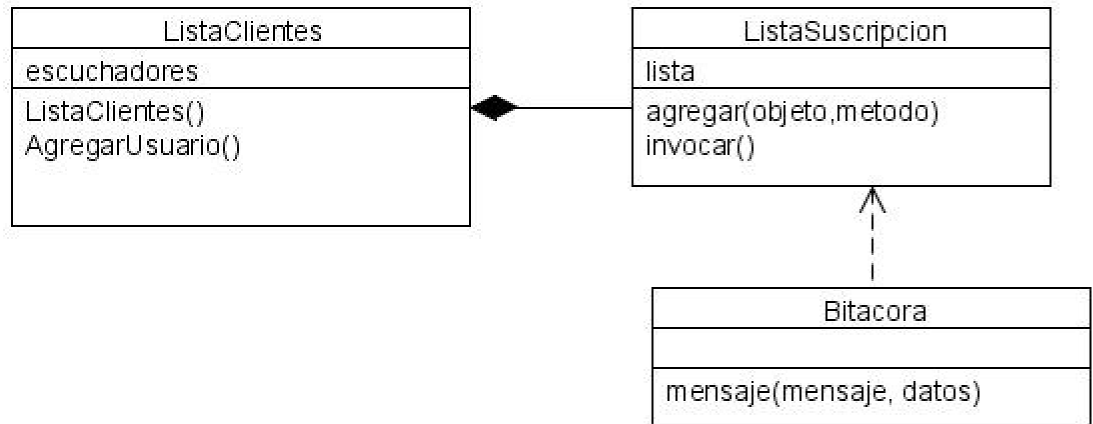
se interesa acerca del estado de la caja y acerca de ser notificado cuando el estado cambia.

En este ejemplo, se demostrará el patrón observer al preparar una lista de clientes observable. Este objeto representa una tabla de base de datos de clientes. El objecto ListaClientes enviará notificaciones cuando nuevos clientes sean añadidos. El objeto utiliza un objeto ListaSuscripcion para implementar la observabiidad. Los objetos que escuchan son una instancia de ListaSuscripcion que otros objetos pueden utilizar para registrarse a sí mismos con ListaClientes.

Los objetos que escuchan usan el método agregar() para añadirse a sí mismos a la lista y ListaClientes utiliza el método invocar() para enviar un mensaje a los que escuchan. No importa si no hay quienes escuchan, o si hay cientos de ellos. Lo especial aquí es que los objetos que escuchan no tienen interacción directa o dependencia de ListaClientes, los que escuchan son aislados de los clientes por la clase ListaSuscripcion.

En el presente ejemplo habrá solo un escuchador. Un objeto bitácora que despliega cualquier mensaje de ListaClientes a la consola. La relación entre los objetos es mostrada en la figura 18

Figura 18. ListaClientes y su ListaSuscripcion con bitácora agregada




```
//EjemploObserver.php
 <?php
// CLASE BITACORA
 class Bitacora
 {
   public function mensaje( $enviador, $TipoMensaje, $datos )
   { print $TipoMensaje." - ".$datos."\n";  }
 }
 class ListaSuscripcion
 {
           var $lista = array();
           public function agregar( $objeto, $metodo )
           { $this->lista []= array( $objeto, $metodo );}

           public function invocar()
           {
// OBTIENE LA LISTA DE ARGUMENTOS DE LA FUNCION
           $args = func_get_args();
```

```
foreach( $this->lista as $l )
// LLAMA A LA FUNCION DE USUARIO CON MATRIZ DE PARAMETROS
{  call_user_func_array( $l, $args );                    }
}
}
class ListaClientes
{
public $escuchadores;

public function ListaClientes()
{ $this->escuchadores = new ListaSuscripcion();}

public function agregarUsuario( $usuario )
{$this->escuchadores->invocar( $this, "agregar", "$usuario" );}
}

$l = new Bitacora();
$cl = new ListaClientes();
$cl->escuchadores->agregar( $l, 'mensaje' );
$cl->agregarUsuario( "starbuck" );
?>
```

5.1.4 Explicación del ejemplo

Al correr el código php el resultado es el siguiente

agregar - starbuck

El código primero hace una bitácora y lista de clientes. Posteriormente, la bitácora se suscribe a la lista de clientes utilizando el método agregar(). El paso final es agregar un usuario a la lista de clientes. La adición del cliente dispara un mensaje al escuchador en este caso, la bitácora que despliega el mensaje acerca de la adición del cliente.

Debería ser fácil extender este código ya sea para hacer algún aprovisionamiento de cliente basado en la adición de cliente o para enviar algún correo, ambos sin cambiar el código en ListaClientes. Esto es aflojar el acoplamiento, y es la razón por la que el patrón observer es tan importante. Existen innumerables usos para el patrón observer en el desarrollo de software. Los sistemas basados en ventanas usan patrones observer y los llaman eventos. Compañías como Tibco manejan su modelo de negocios entero vía el patrón observer, conectando grandes sistemas de negocios como recursos humanos y planilla. Sistemas de base de datos usan un patrón observer y llaman a código que escuchan eventos de triggers. Estos triggers son activados cuando ciertos tipos de registros son cambiados en la base de datos. Un enfoque con patrón observer es también útil siempre que se piense que un cambio de estado es relevante pero no se entiende para quién será relevante; se pueden codificar los escuchadores más tarde y no atarlos al objeto que será observado.

Un potencial error con el patrón observer es el ciclo infinito. Esto puede suceder cuando elementos que observan un sistema pueden también alterar tal sistema. Por ejemplo, un combo desplegable altera un valor y muestra la estructura de datos acerca de él. Esa estructura de datos entonces notifica al combo desplegable que el valor ha cambiado, con lo cual el combo cambia su valor para coincidir, solo para enviar otra notificación a la estructura de datos, y

así sucesivamente. La forma más fácil de resolver este problema es codificar el combo de tal manera que se prevenga la recursión. Esto simplemente ignoraría un mensaje de la estructura de datos si está actualmente en medio de notificar a la estructura de datos acerca de un nuevo valor.

5.2 Uso del patrón de diseño factory

En la programación orientada a objetos, la forma más común de hacer un objeto es con el operador new, el constructor de lenguaje para tal fin. Pero en algunos casos new puede ser problemático. Por ejemplo, al hacer muchas clases de objetos requiere una serie de pasos: se podría necesitar computar o traer la configuración inicial del objeto; o se habría que tener que escoger a cuál instanciar de muchas clases; o quizás se tiene que hacer un lote de otros objetos que ayuden antes de hacer el objeto que se necesita. En esos casos new es un “proceso” más que una operación, un diente de una rueda de maquinaria.

5.2.1 El problema

Cómo se puede hacer tales objetos “complejos” fácil y convenientemente, sin programar copiando y pegando.

5.2.2 La solución

Hacer un “factory”, una función o método de clase para “fabricar” nuevos objetos. Para entender el valor de un factory, piénsese en el diferencia entre

```
$connexion =& new MySqlConnection($usuario, $password, $basedatos);
```

….Disperso a lo largo del código, y la más concisa…..

```
$connexion =& hacer_conexion();
```

El último pedazo de código centraliza el mismo para hacer una conexión a base de datos en la fábrica o factory hacer_conexion(), y siguiendo la analogía anterior, transforma el proceso de hacer una conexión de base de datos a una simple operación, una operación justo como new. El patrón factory inyecta "inteligencia" a la creación del objeto. Encapsula la creación de un objeto y retorna el nuevo objeto al que lo llamó.

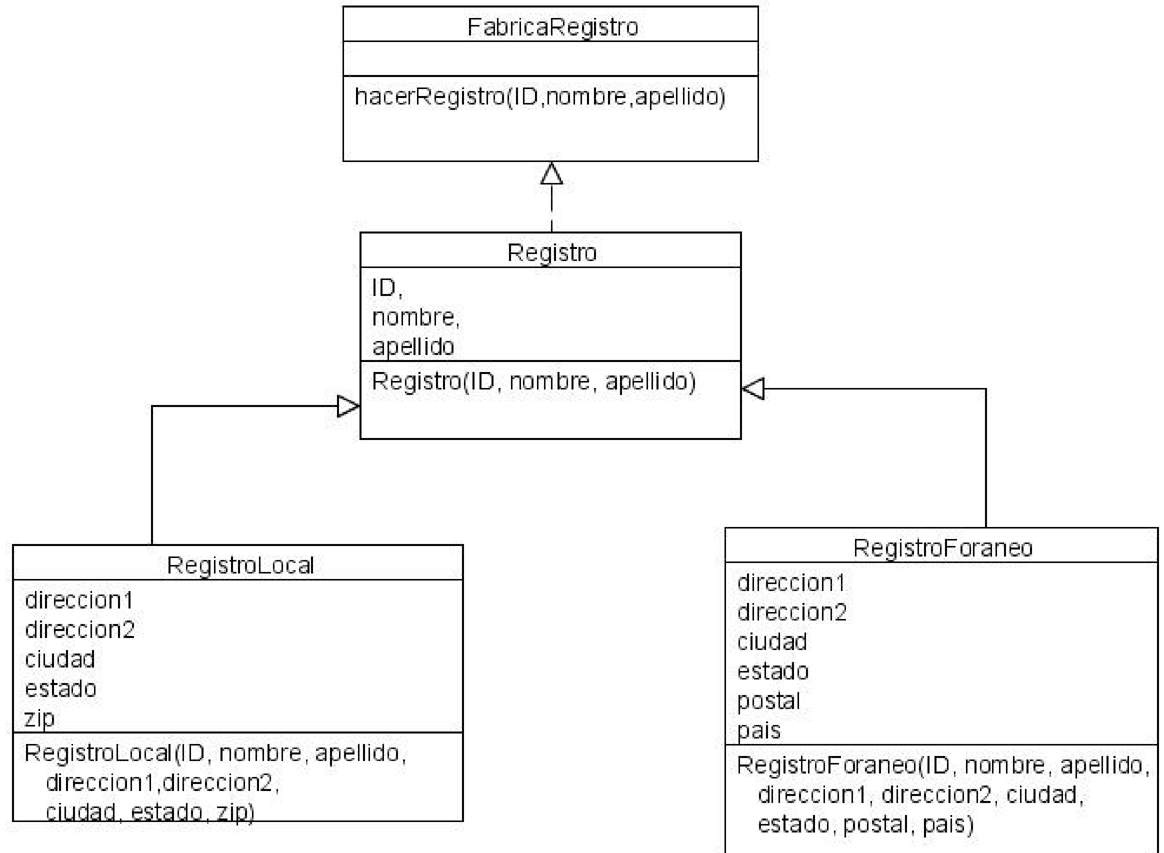
Se necesita cambiar la estructura de un objeto y cómo es creado. Solo hay que ir al factory del objeto y cambiar el código una vez. El patrón factory es tan útil porque es fundacional, es decir, que aparece una vez y otra vez en muchos otros patrones y aplicaciones.

5.2.3 Código de ejemplo

El patrón abstracto factory es la máquina expendedora de los patrones de diseño. Se solicita lo que se desea, y como una máquina expendedora vende un objeto basado en el criterio. El valor es que se puede cambiar qué tipos de objetos son creados a través del sistema al solamente alterar el factory.

En este ejemplo el factory hace objetos Registro, donde cada registro tiene un ID, un nombre y un apellido. La relación entre clase se muestra en la figura 19.

Figura 19. Las clases Registro y FabricaRegistro



No hay manera para forzar estrictamente que solo el factory pueda hacer objetos de un tipo particular en PHP. Pero si se usa el factory lo suficientemente a menudo, las ingenieros que copian y pegan el código terminarán usando el factory, se convertirá rápidamente la forma de facto de hacer diferentes tipos de objeto.

El código

```
<?php
   class Registro
   {
```

```
  public $id = null;
  public $nombre = null;
  public $apellido = null;

  public function __construct( $id, $nombre, $apellido )
  {
     $this->id = $id;
     $this->nombre = $nombre;
     $this->apellido = $apellido;
  }
}

class RegistroLocal extends Registro
{
   public $direccion1 = null;
   public $ direccion2 = null;
   public $ciudad = null;
   public $estado = null;
   public $zip = null;

   public function __construct( $id, $nombre, $apellido,
      $direccion1, $ direccion2, $ciudad, $estado, $zip )
   {
      parent::__construct( $id, $nombre, $apellido );
      $this->direccion1 = $direccion1;
      $this-> direccion2 = $addr2 direccion2
      $this->ciudad = $ciudad;
      $this->estado = $estado;
      $this->zip = $zip;
```

```
    }
 }

 class RegistroForaneo extends Registro
 {
     public $direccion1 = null;
    public $direccion1 = null;
    public $ciudad = null;
    public $estado = null;
    public $postal = null;
    public $pais = null;

    public function __construct( $id, $nombre, $apellido,
       $direccion1, $direccion2, $ciudad, $estado, $postal, $pais )
    {

       parent::__construct( $id, $nombre, $apellido );
       $this-> direccion1= $direccion1;
       $this-> direccion2= $direccion2;
       $this->ciudad = $ciudad;
       $this-> estado = $estado;
       $this-> postal = $postal;
       $this-> pais = $pais;
    }
}

class FabricaRegistro
{
```

```
public static function hacerRegistro( $id, $nombre, $apellido,
   $direccion1, $direccion2, $ciudad, $estado, $postal, $pais )
{
    if ( strlen( $pais ) > 0 && $pais != "USA" )
       return new RegistroForaneo( $id, $nombre, $apellido,
           $direccion1, $direccion2, $ciudad, $estado, $postal, $pais );
    else
        return new RegistroLocal( $id, $nombre, $apellido,
            $direccion1, $ direccion2, $ciudad, $estado, $postal );
    }
}

function leerRegistros()
{
    $registros = array();

    $registros []= FabricaRegistro::hacerRegistro(
        1, "Jack", "Herrington", "4250 San Jaquin Dr.", "",
        "Los Angeles", "CA", "90210", ""
    );
    $records []=FabricaRegistro:: hacerRegistro (
         1, "Megan", "Cunningham", "2220 Toorak Rd.", "",
         "Toorak", "VIC", "3121", "Australia"
        );
        return $records;
}
$registros = leerRegistros();
foreach( $registros as $r )
{
```

```
        $clase = new ReflectionClass( $r );
        print $clase->getName()." - ".$r->id." - ".$r->nombre." - ".$r->apellido."\n";
    }
?>
```

5.2.4 Explicación del ejemplo

La primera sección del código implementa la clase base Registro, así como las clases derivada RegistroLocal y RegistroForaneo. Estas son envoltorios de datos bastante simples. Entonces la clase factory puede construir ya sea un RegistroLocal o un RegistroForaneo dependiendo de la data que se le pase. El código de prueba añadido al final del script añade unos pocos registros, y entonces imprime su tipo junto con algo de sus datos. Al corrrer el script de una página php guardada como EjemploFactory.php se obtiene el siguiente resultado:

```
RegistroLocal - 1 - Jack - Herrington
RegistroForaneo - 1 - Megan – Cunningham
```

Se puede utilizar el patron abstract factory en una aplicación PHP de base de datos de diferentes maneras:

Creación de objetos de bases de datos: la factory provee cualquiera de los tipos asociados con las diferentes tablas en la base de datos.

Creación de objetos portable: la factory provee una cantidad de objetos diferentes dependiendo ya sea del tipo de sistema operativo el código se correo o las diferentes bases de datos que la aplicación tiene adjunta.

Creación por estándar: la aplicación soporta varios estándares de formato de archivo y usa la factory para hacer un objeto apropiado para el tipo de archivo dado. Los lectores de archivo pueden registrarse a sí mismos con el factory para añadir soporte sin tener que cambiar a cualquiera de los clientes. Después de utilizar los patrones por un tiempo, se puede desarrollar un sentido de cuándo es sensato utilizar un tipo particular de patrón. Se podría utilizar este patrón cuando se está haciendo un feo lote de construcciones con varios tipos de objetos. Se encontrará que si se necesita cambiar los tipos de objetos que son creado, o cómo ellos son creados, se tendrá mucho código para cambiar. Si se usa un factory, se necesitará cambiar esa creación de objeto en un solo lugar.

5.3 Uso del patrón de diseño strategy

Cuando se desarrolla código orientado a objetos, se necesita a veces que un objeto cambie su comportamiento ligeramente basado en circunstancias. Por ejemplo, un menú podría presentarse a sí mismo horizontal o verticalmente dependiendo de la preferencia de diseño de un usuario, o un pedido podría calcular impuesto sobre ventas de forma diferente basado en la dirección de envío del cliente.

Una típica implementación de un objeto como Menu tiene métodos para agregar(), eliminar() y reemplazar() elementos de menú, setear() el estilo, y mostrar() a sí mismo. No importa qué clase de menú se quiera hacer, los menús ofrecen una interfase consistente; solo los algoritmos internos de uno o más métodos -al menos mostrar(), por ejemplo- difieren. Pero, ¿qué sucede digamos, cuando el número de estilos de menús se expande?, o en el caso del pedido, ¿qué ocurre cuando departamento, estado y las reglas extranjeras de

impuestos son tomadas en cuenta? Si muchos métodos tienen instrucciones "case" para casos especiales, una simple encapsulación de otro caso pronto resulta complicada, difícil de leer y difícil de mantener.

5.3.1 El problema

¿Cómo se puede cambiar la implementación interna de un objeto fácilmente, escogiendo una implementación para utilizar a la vez que el script es ejecutado, en lugar de cuando es escrito?. ¿Cómo se puede codificar un conjunto de implementaciones que sean fácil de mantener y extender?

5.3.2 La solución

Cuando una clase abarca múltiples implementaciones y una instancia puede dinámicamente escoger cualquiera de esas implementaciones, se usa el patrón strategy para separar el objeto de sus algoritmos. O, para ponerlo más simple, si los métodos de una clase utilizan instrucciones "case" constantemente, es un buen candidato para refactorizar dentro del patrón strategy.

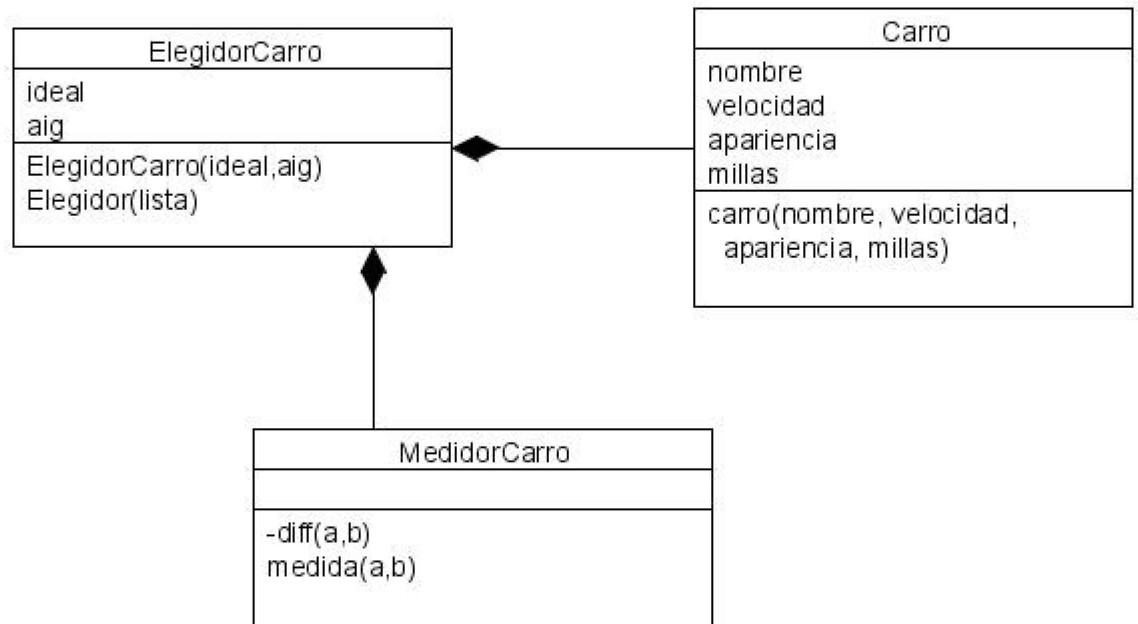
El patrón strategy es muy poderoso debido a que la idea esencial del mismo es el pincipio orientado a objetos del polimorfismo. Hay claros ejemplos del patrón strategy fuera del dominio de la programación. Si necesito ir de la casa al trabajo en la mañana, pueden escoger entre varias estrategias: puedo manejar mi carro, tomar el bus, ir en bicicleta o volar en helicóptero. Cada estrategia tiene los mismos resultados pero usa recursos de forma diferente, y la opción de la estrategia depende del tiempo consumido, la disponibilidad de recursos particulares (como tener un vehículo) y la conveniencia de cada

método. Una buena estrategia en un día puede ser una pobre estrategia al siguiente, así que la opción de la estrategia tiene que se hecha dinámicamente.

5.3.3 Código de ejemplo

En este ejemplo, se utilizará un carro. Este código recomendará un carro basado en algunos criterios de búsqueda. En este caso, se darán especificaciones para un carro ideal y se dejará que el código traiga el carro que más cercanamente coincide con las expectativas deseadas. El valor del patrón strategy es que se puede alterar el código de comparación de carro independientemente de la selección del mismo. En la figura 12 se muestra el UML. El objeto ElegidorCarro usa un objeto MedidorCarro para comparar cada carro con el modelo ideal. Entonces, el mejor carro es retornado al cliente.

Figura 20. Relación entre objetos ElegidorCarro, MedidorCarro y Carro



Código

```
<?php
//
  class Carrro
  {
    public $nombre;
    public $velocidad;
    public $apariencia;
    public $millas;
    public function Carro( $nombre, $velocidad, $apariencia, $millas )
    {
      $this->nombre = $nombre;
      $this->velocidad = $velocidad;
      $this->apariencia = $apariencia;
      $this->millas = $millas;
    }
  }

  class MedidorCarro
  {
    private function diff( $a, $b )
    {
      return abs( $a - $b );
    }

    public function medida( $a, $b )
    {
      $d = 0;
```

```
    $d += $this->diff( $a->velocidad, $b->velocidad);
    $d += $this->diff( $a->apariencia, $b-> apariencia);
     $d += $this->diff( $a->millas, $b-> millas);
     return ( 0 - $d );
  }
}

class ElegidorCarro
{
  private $ideal;
  private $alg;

  function ElegidorCarro( $ideal, $alg )
  {
    $this->ideal = $ideal;
    $this->alg = $alg;
  }

  public function elegir( $lista )
  {
    $minrank = null;
    $encontrado = null;
    $alg = $this->alg;

    foreach( $lista as $carro )
    {
      $rank = $alg->medida( $this->ideal, $carro );
      if ( !isset( $minrank ) ) $minrank = $rank;
      if ( $rank >= $minrank )
```

```
    {
        $minrank = $rank;
        $encontrado= $car;
    }
  }

  return $encontrado;
  }
}

function llevarCarro( $carro )
{
  $lista = array();
  $ lista [ ]= new Carro( "zippy", 90, 30, 10 );
  $ lista [ ]= new Carro( "mom'n'pop", 45, 30, 55 );
  $ lista [ ]= new Carro( "beauty", 40, 90, 10 );
  $ lista [ ]= new Carro( "enviro", 40, 40, 90 );

  $cw = new MedidorCarro();
  $cc = new ElegidorCarro( $car, $cw );
  $encontrado = $cc->elegir( $lista );
  echo( $encontrado->name."\n" );
}

llevarCarro( new Carro( "ideal", 80, 40, 10 ) );
llevarCarro ( new Carro( "ideal", 40, 90, 10 ) );
?>
```

5.3.4 Explicación del ejemplo

Comenzando desde el principio del archive, se define la clase Carro, la cual guarda el nombre del mismo y las métricas de velocidad, apariencia y millas. Cada una está calificada de 0 a 100 (para hacer el cálculo fácil). Entonces viene la clase MedidorCarro, la cual compara dos carros y regresa una métrica de comparación. Finalmente, está la clase ElegidorCarro, la cual usa una clase MedidorCarro para seleccionar el mejor carro basado en algunos criterios de entrada. La función llevarCarro() hace un conjunto de carros y entonces usa una clase ElegidorCarro para tomar el carro de la lista que mejor se ajusta al criterio (pasado vía otro objeto Carro).

Al correr el código de PHP de prueba guardado como EjemploStrategy.php se obtiene el siguiente resultado:

```
% php strategy.php
zippy
beauty
```

La salida muestra que el carro recomendado si se desea velocidad es el “zippi”, una buena aproximación. El carro recomendado si se desea algo más sexi es l carro “beauty”.

El código que deduce que un carro es una buena elección está totalmente abstraído del código que atraviesa la lista de carros y trae uno de esa lista. Se puede cambiar el algoritmo que mide un cierto carro independientemente del código que lo trae el cual es el carro correcto de la lista medida. Por ejemplo, se pueden agregar carros que le han interesado recientemente o que se han poseído en el pasado dentro del algoritmo de

medición. O se puede cambiar el código de llevarCarro para seleccionar los tres mejores y proveer una elección entre ellos.

5.4 Uso del patrón de diseño adapter

Las interfaces cambian. Es un simple y perenne hecho que los programadores tienen que aceptar y lidiar. Los vendedores cambian su código; las librerías de sistema son revisadas; y los lenguajes de programación y sus librerías evolucionan.

5.4.1 El problema

¿Cómo se puede proteger a sí mismo de cambios en el API de librerías externas que se utilizan?. Si se escribe una librería, ¿se puede proveer un medio para permitir a usuarios existentes actualizarse suavemente, aún si se ha cambiado el API?. ¿Cómo se puede cambiar la interfase de un objeto para que se ajuste mejor a las necesidades?

5.4.2 La solución

El patrón adapter provee una interfase diferente a un objeto. Se puede utilizar adapter para realizar una interfase familiar a un objeto diferente, evitando el fastidio de actualizar o refactorizar el código cliente.

Hay que considerar qué ocurre cuando el API de la librería de un tercero cambia. Se podría solamente morder el polvo y cambiar todo el código cliente, pero a menudo no es tan simple. Se podría estar trabajando en un nuevo proyecto que requiere las características de la nueva versión de la librería, pero ya tiene actualmente docenas de antiguas aplicaciones legadas que funcionan

bien con la versión previa de la librería. Probablemente no se podría justificar el uso de la nueva característica si la actualización significa tocar el código cliente para todas las otras aplicaciones también.

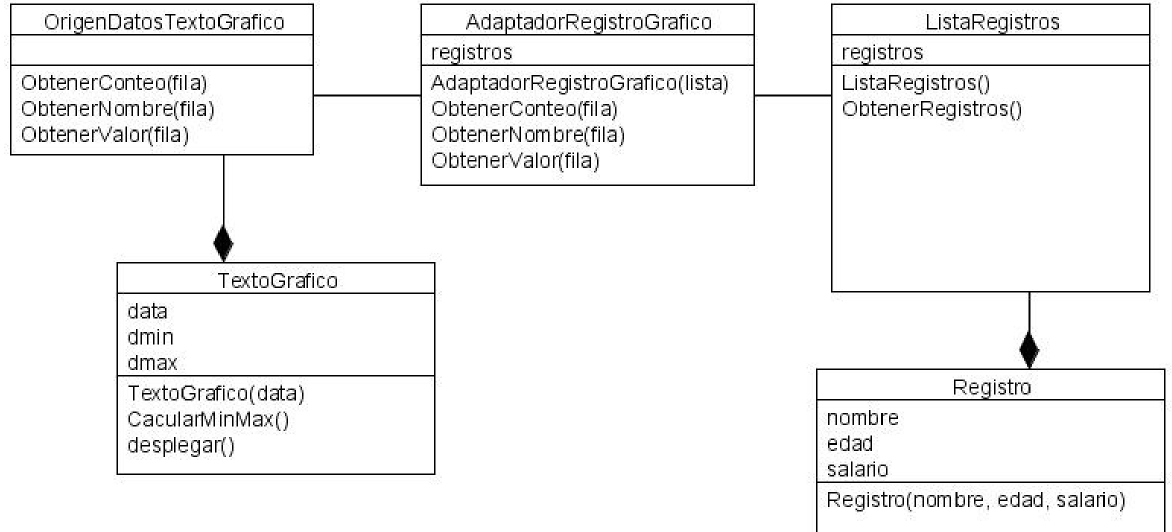
5.4.3 Código de ejemplo

Se utilizará una clase adaptador para transferir datos entre dos módulos cuando no se quiere cambiar el API de cualquiera de ellos. Algunas veces se tiene que obtener datos de dos objetos, cada uno de los cuales utiliza un formato diferente de datos. Cambiar uno o ambos objetos simplemente no es una opción debido a que se tendrá que hacer toda clase de otros cambios en el resto del código. Una solución a este problema es utilizar una clase adaptador. Un adaptador es una clase que entiende ambos lados de la barrera de transferencia de datos y adapta un objeto para hablar con otro.

El adaptador demostrado en este ejemplo adapta datos de un objeto de base de datos en data usable por medio de un motor de texto y gráfico.

La figura 21 muestra al AdaptadorRegistroGrafico situado entre el TextoGrafico en la izquierda y la ListaRegistros en la derecha. El objeto TextoGrafico especifica muy bien el formato que espera para datos utilizando una clase abstracta llamada OrigenDatosTextoGrafico. La ListaRegistros es una clase contenedora que tiene una lista de registros, donde cada registro contiene un nombre, edad y salario.

Figura 21. El adaptador situado entre la gráfica y los datos



Para este ejemplo, se desea una gráfica de los salarios. El trabajo del adaptador es tomar datos de ListaRegistros y convertirlos en una forma ajustable para TextoGrafico al cambiar los datos dentro de un objeto DatosOrigenTextoGrafico.

El código

```
<?php
   abstract class OrigenDatosTextoGrafico
   {
      abstract function ObtenerConteo();
      abstract function ObtenerNombre( $fila );
      abstract function ObtenerValor( $fila );
   }

   class TextoGrafico
```

```
{
    private $data;
    private $dmin;
    private $dmax;
    public function TextoGrafico( $data )
    {
        $this->data = $data;
    }

    protected function CalcularMinMax()
    {
        $this->dmin = 100000;
        $this->dmax = -100000;
        for( $r = 0; $r < $this->data->ObtenerConteo(); $r++ )
        {
            $v = $this->data->ObtenerValor( $r );
            if ( $v < $this->dmin ) { $this->dmin = $v; }
            if ( $v > $this->dmax ) { $this->dmax = $v; }
        }
    }

    public function desplegar()
    {
        $this->CalcularMinMax();
        $ratio = 40 / ( $this->dmax - $this->dmin );
        for( $r = 0; $r < $this->data->obtenerConteo(); $r++ )
        {
            $n = $this->data->ObtenerNombre( $r );
            $v = $this->data->ObtenerValor( $r );
```

```
        $s = ( $v - $this->dmin ) * $ratio;
        echo( sprintf( "%10s : ", $n ) );
        for( $st = 0; $st < $s; $st++ ) { echo("*"); }
        echo( "\n" );
    }
  }
}

class Registro
{
    public $nombre;
    public $edad;
    public $salario;
    public function Registro( $nombre, $edad, $salario )
    {
        $this->nombre = $nombre;
        $this->edad = $edad;
        $this->salario = $salario;
    }
}

class ListaRegistros
{
    private $registros = array();

    public function ListaRegistros()
    {
        $this->registros []= new Registro( "Jimmy", 23, 26000 );
        $this-> registros []= new Registro ( "Betty", 24, 29000 );
```

```
        $this-> registros []= new Registro ( "Sally", 28, 42000 );
        $this-> registros []= new Registro ( "Jerry", 28, 120000 );
        $this-> registros []= new Registro ( "George", 43, 204000 );
    }

    public function ObtenerRegistros()
    {
        return $this->registros;
    }
}

class AdaptadorRegistroGrafico extends OrigenDatosTextoGrafico
{
    private $registros;

    public function AdaptadorRegistroGrafico( $rl )
    {
      $this->registros = $rl->ObtenerRegistros();
    }
    public function ObtenerConteo( )
    {
        return count( $this->registros );
    }
    public function ObtenerNombre( $fila )
    {
        return $this->registros[ $fila ]->nombre;
    }
    public function ObtenerValor( $fila )
    {
```

```
            return $this->registros[ $fila ]->salario;
        }
    }

    $rl = new ListaRegistros();
    $ga = new AdaptadorRegistroGrafico( $rl );
    $tg = new TextoGrafico( $ga );
    $tg->desplegar();
?>
```

5.4.4 Explicación del ejemplo

La porción superior del archivo es dedicado al gráfico. Esta porción define la clase abstracta OrigenDatosTextoGrafica así como también la clase TextoGrafica. TextoGrafica utiliza un OrigenDatosTextoGrafica para referenciar los datos. La sección media define el Registro y la ListaRegistros, la cual guarda los datos a ser graficados. La tercera sección define el AdaptadorRegistroGrafico, el cual adapta la ListaRegistros a un origen utilizable por la gráfica.

El código al final primero hace una ListaRegistros y entonces hace el adaptador y el TextoGrafico, con una referencia al adaptador. La gráfica despliega los datos al leer los datos del adaptador.

Al correr al código guardado como EjemploAdapter.php se obtiene el siguiente resultado:

```
% php adapter.php
Jimmy :
```

Betty : *

Sally : ****

Jerry : **********************

George : **

El final de debajo de la escala es Jimmi, y el máximo el George. La gráfica automáticamente lo escala de tal manera que Jimmy es mostrado sin estrellas (mínimo), y George con 40 estrellas (el máximo). Más importante, por supuesto, este código manejó la conversión de datos sin problema, y sin hacer desastres con el código interno de la clase Registro. Se puede usar un adapter siempre que se tengan dos APIS que necesiten trabajar juntos, y donde cambiar cualquiera de esos APIS no es una opción.

5.5 Uso del patrón singleton

En casi cada programa orientado a objetos, hay usualmente uno o dos recursos que son creados una vez y compartidos durante la aplicación entera. Por ejemplo una conexión a base de datos en una aplicación e-commerce es uno de tales recursos: es inicializada cuando la aplicación se lanza, es utilizada para efectuar todas las transacciones y es finalmente desconectada y destruida cuando el programa termina. En el código, no hay necesidad de llamar a una conexión de base de datos a cada momento; es fastidioso y muy ineficiente. En lugar de eso, el código puede simplemente reutilizar la conexión que ya ha sido establecido. El reto entonces es cómo se refiere a la conexión (o a cualquier otro único recurso perenne, tal como un archivo abierto o una cola).

5.5.1 El problema

¿Cómo se puede asegurar que una instancia de una clase particular es exclusiva (es siempre la sola instancia de esa clase) aunque también sea fácilmente accesible?.

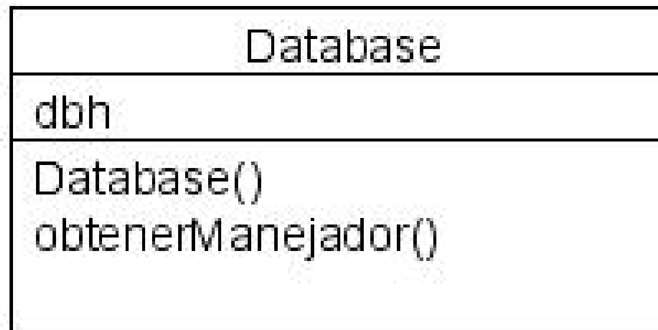
5.5.2 La solución

Por supuesto, una variable global es una solución obvia, pero también es una caja de pandora (el dicho "el buen juicio viene de la experiencia, pero la experiencia usualmente viene de un pobre juicio" viene a la mente). Cualquier porción del código puede modificar una variable global, causando sin fin de depuraciones que se agravan, cualquier cantidad de problemas inesperados. En otras palabras el estado de una variable global es siempre cuestionable. Cuando se necesita una instancia exclusiva de una clase particular, se usa el patrón acertadamente llamado singleton. Una clase basada en el patrón singleton instancia e inicializa apropiadamente una instancia de la clase y provee acceso al mismo objeto exacto, típicamente a través de un método estático.

5.5.3 Código de ejemplo

Un singleton, como se dijo antes, es un objeto que tiene solo una instancia a cualquier momento en el sistema. Un gran ejemplo de un potencial singleton es una manejador de base de datos. Para cada instancia del intérprete de PHP, debería haber solo un manejador de base de datos. Este ejemplo implemente tal como una configuración, una versión de singleton de un manejador de base de datos.

Figura 22. El singleton de base de datos



Es bastante simple, el objeto contiene el manejador de base de datos y tiene dos métodos. El primero es el constructor, el cual es privado para asegurarse que nadie afuera de clase pueda crear el objeto. También contiene un método estático llamada obtenerManejador que retorna el manejador de base de datos.

Código:

```
<?php
  require( 'DB.php' );
  class Database
  {
    private $dbh;
  private function Database()
  {
     $dsn = 'mysql://root:password@localhost/test';
     $this->dbh =& DB::Connect( $dsn, array() );
     if (PEAR::isError($this->dbh)) { die($this->dbh->getMessage()); }
```

```
    }

    public static function obtenerManejador()
    {
         static $db = null;
        if ( !isset($db) ) $db = new Database();
            return $db->dbh;
        }
    }
   echo( Database::obtenerManejador ()."\n" );
   echo( Database:: obtenerManejador ()."\n" );
   echo( Database:: obtenerManejador ()."\n" );
?>
```

Este simple singleton tiene un constructor que entra en la base de datos y un método estático que accesa y crea un objeto si no ha sido creado ya retornando el manejador de base de datos de tal objeto. Si se usa este método para obtener manejadores de base de datos, se puede descansar tranquilo asegurado que se conectará a la base de datos una sola vez por página que trae datos.

Al correr el script en el navegador, se obtiene lo siguiente:

```
Object id #2
Object id #2
Object id #2
```

Esto demuestra que las múltiples llamadas al método estático obtenerManejador() están retornando el mismo objeto una y otra vez. Esto

significa que cada vez que la llamada fue hecha, el objeto Database, y por lo tanto el manejador de base de datos, fue utilizado.

5.5.4 Segundo ejemplo

Manejadores de bases de datos son una cosa, pero, ¿qué acerca de algo más completo?. En este ejemplo se probará una lista compartida de estados, como se muestra en el siguiente código:

```
<?php
  class ListaEstados
  {
     private $estados = array();
     private function ListaEstados()
     {
     }

     public function agregarEstado( $estado )
     {
        $this-> estados[]= $estado;
     }

     public function obtenerEstados()
      {
       return $this-> estados;
      }

      public static function instance()
      {
```

```
            static $estados = null;
            if ( !isset($estados) ) $estados = new ListaEstados();
            return $estados;
          }
        }

        ListaEstados::instance()->agregarEstado( "Florida" );
        var_dump( ListaEstados::instance()->obtenerEstados( ) );

        ListaEstados::instance()->agregarEstado( "Kentucky" );
        var_dump( ListaEstados::instance()->obtenerEstados( ) );
    ?>
```

5.5.5 Explicación del segundo ejemplo

Este código hace una clase singleton, ListaEstados, la cual contiene una lista de estados. Se pueden agregar estados a la lista, así como obtener un listado de los estados. Para acceder a la sola instancia compartida de este objeto, se tiene que usar el método estático instante() (en lugar de hacer una instancia directamente). Al correr el script en el navegador da el siguiente resultado:

```
% php singleton2.php
   array(1) {
     [0]=>
     string(7) "Florida"
   }
   array(2) {
     [0]=>
```

```
    string(7) "Florida"
    [1]=>
    string(8) "Kentucky"
  }
```

El primer volcado de variable muestra que justamente el primer estado, Florida, está en la lista. El segundo volcado muestra la adición de Kentucky al objeto compartido. Algunos autores son vacilantes a la hora de recomendar el patrón singleton demasiado por considerar está sobreutilizado, al observar códigos que tienen unos rodeos bastante feos para lidiar con objetos singleton, lo cual refleja que el patrón singleton está siendo utilizado incorrectamente. Si se están tomando pasos significativos para trabajar con un singleton, podría no ser un uso apropiado del patrón.

CONCLUSIONES

1. La identificación, descripción, recuperación, adaptación e integración de activos de software son todos asuntos importantes. Lograr algún progreso en cualquiera de ellos será un paso hacia la reutilización efectiva.

2. El área completa de la reutilización de software es más amplia: no sólo abarca diseño, sino el ciclo de vida completo y no sólo relaciona el diseño con la reutilización, sino también diseño para la reutilización.

3. El sólo uso de tecnologías orientadas a objetos no darán una reutilización de software efectiva. Las tecnologías son sólo una pequeña parte de las prácticas de reutilización, y si se adoptan lenguajes de programación orientados a objetos (ej. Java), sin implementar otros pasos involucrados en un programa de reutilización quedarán decepcionados.

4. Con catálogos de soluciones típicas, metodologías estandarizadas, lenguajes, etc., se es testigo de una disciplina emergente de ingeniería; y los prospectos de esto son bastantes promisorios, a pesar de la longitud del camino que todavía queda.

5. La situación hoy en día con los patrones de diseño orientados a objetos es que ya existen catálogos. Usarlos y aplicarlos no parece ser una tarea fácil. Los patrones pueden ser difícilmente entendidos completamente en la primera lectura. Un diseñador tendrá que referirse a ellos una vez y otra vez.

6. Ser capaz de aplicar un patrón exitosamente requiere conocimiento acerca del patrón, comprenderlo y examinarlo. Después de eso, los patrones de diseño pueden ser aplicados repetidamente por analogía.

7. Los recientes esfuerzos en torno a la reutilización deberían ser considerados pasos en la dirección correcta. Si la disciplina de desarrollo de software es para cambiar de arte a ingeniería, entonces la reutilización de software es definitivamente uno de los mayores factores influenciando el cambio.

RECOMENDACIONES

1. Proponer un formalismo para representación de patrones de diseño, marcos de trabajo y arquitecturas de software que servirían primariamente para recuperarlos y aplicarlos en diseño de software.

2. Investigar y desarrollar herramientas que apoyarían al usuario en recuperar y aplicar patrones de diseño, marcos y arquitecturas de software en diseño de software.

3. En el caso de los patrones de diseño se recomienda adaptar el patrón de diseño al problema y no el problema al patrón de diseño.

4. Investigar las posibilidades de incorporar patrones de diseño, marcos de trabajo y arquitecturas de software dentro de un lenguaje de diseño/implementación.

BIBLIOGRAFÍA

1. Buschmann Frank y otros. “Pattern-Oriented Software Architecture: ASystem of Patterns”. Wiley, 1996.

2. Clements Paul y Northrop Linda. “Software Product Lines: Practices and Patterns”. Addison-Wesley, 2002.

3. Gamma E. y otros. “Design Patterns, Elements of Reusable Object-Oriented Software”. Addison-Wesley,1995.

4. Herrington Jack, “PHP Hacks”. O'Reilly, 2005.

5. Jacobson Ivar, Griss Martin y Jonsson Patrik. “Software Reuse: Architecture, Process and Organization for Business Success”. Addison-Wesley, 1997.

6. Sweat Jason E., “php|architect’s Guide to PHP Design Patterns”. Marco Tabini & Associates, 2005.

BIBLIOGRAFÍA ELECTRÓNICA

7. Müller Hausi A.. “Understanding software systems using reverse engineering technologies research and practice” (tutorial). En la 18 Conferencia Internacional de Ingeniería de Software *(ICSE-18)*, Berlin, Alemania, Marzo 1996. Disponible en : http://www.rigi.csc.uvic.ca/UVicRevTut/UVicRevTut.html.

8. IEEE Computer Society. “Software Reuse: A Standards Based Guide”. Disponible en línea en: http://www.computer.org/cspress/catalog/bp00874.htm

9. Software Engineering Institute, Carnegie-Mellon University. “DomainEngineering: A Model-Based Approach”, Enero 2000. Disponible en: http://www.sei.cmu.edu/domain-engineering/.

10. Software Engineering Institute, Carnegie-Mellon University. “Domain Engineering and Domain Analysis”, Septiembre 2000. Disponible en línea en: http://www.sei.cmu.edu/str/descriptions/deda-body.html.

11. http://www.objectmentor.com/resources/publishedArticles.html