UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

U. A. C. P. y P.

DESARROLLO DE UN SISTEMA DIDACTICO DE PATRONES DE DISEÑO.

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

JORGE FLORES MALDONADO

ASESOR DE TESIS: DRA. HANNA OKTABA

MEXICO. D. F. SEPTIEMBRE DE1998.

Patrones de Diseño

Autor: Jorge Flores Maldonado
Asesor: Dra. Hanna Oktaba
IIMAS-UNAM
Maestría en Ciencias de la Computación

Enero 18, 1998

Indice

Reconocimientos

*B*astantes personas han participado en forma directa e indirecta en la elaboración, investigación, apoyo técnico y administrativo para la elaboración de este trabajo. En especial mi reconocimiento a las siguientes personas: Dra. Hanna Oktaba por haberme inducido en esta interesante área de la orientación a objetos. Lulu, Juanita, Violeta, Alfredo, Lolita, Gaby, y Mardonio del equipo administrativo y de intendencia. Dante, Mary y demás compañeros por las constantes preguntas de caracter técnico y metodológico. A los profesores de la maestría en ciencias de la computación del IIMAS-UNAM.

Debo hacer patente que sin el apoyo de la beca estudiantil otorgada por el CONACYT no me hubiese sido posible ser estudiante de tiempo completo y mucho menos dedicarme a la investigación, desarrollo e implementación de este trabajo, que se presenta como tesis para sustentar el grado de maestro de la maestria en ciencias de la computación del IIMAS-UNAM.

Por último a mis padres y mi novia Montserrat por su apoyo moral y afectivo en esta ardua pero excitante labor de investigación.

Prefacio

En la comunidad de desarrolladores de software, los patrones están recibiendo bastante atención desde la publicación del libro Design Patterns [*3*] de los autores Gamma, Helm, Johnson y Vlissides en 1995. Los 24 patrones que comprenden el catálogo de patrones de diseño, nombre más común por el cual se le conoce, han promovido el desarrollo de una área especial de la ingeniería de software: Patrones Orientados a Objetos.

El interés en los patrones, radica en el potencial que ofrecen para registrar la experiencia de desarrolladores expertos de software. No sólo representan el conocimiento de la solución a un problema en un contexto específico, sino también trucos, consejos, ventajas, desventajas, identificación de componentes y definición del ámbito de aplicación.

Originalmente, más de la mitad de los patrones del catálogo aparecen publicados en la tesis doctoral de de E. Gamma [*35*], que realizó una investigación exhaustiva buscando soluciones a problemas recurrentes en frameworks (ver *glosario*) y aplicaciones orientadas a objetos a gran escala.

La idea de registrar el conocimiento del diseño en una forma canónica se remonta más allá de la tesis doctoral de E. Gamma. Sus raíces se encuentran en una rama ajena a la ingeniería de software, la arquitectura, con el que se considera el padre de la teoría de patrones de C. Alexander [*16-19*]. Alexander plantea que las construcciones de viviendas comparten similitudes susceptibles de estandarizarse. Sus observaciones comprenden también otras áreas de estudio como la artesanía, en donde descubre que en la elaboración artesanal de alfombras, de la cual es un coleccionador, constantemente se repiten patrones de hilado y tejido. No obstante, la fabricación de productos sigue procedimientos similares, y aunque a primera vista los productos terminados sean semejantes, cada uno es diferente, debido al valor subjetivo y en otros casos objetivo que le imprime su propietario, a lo cual Alexander le llama "quality without a name". Por tanto, bajo un esquema de similitudes y de personalización propia por parte de cada individuo, es posible hablar en términos de un lenguaje de patrones.

El objetivo de este trabajo es retomar las ideas esenciales de Alexander e implementar 7 de los 24 patrones del catálogo de patrones de diseño en un caso de estudio: "el ensamble de un jean". El esquema que se ha seguido es el siguiente:

- La primera parte comprende varios puntos: el primero es una breve historia de los patrones en dos áreas, la arquitectura y la ingeniería de software; el segundo comprende varias definiciones de los autores más citados, posteriormente se elabora una definición que sirve como guía en el desarrollo de todo el trabajo; en el tercer punto se propone una clasificación de los patrones en tres vertientes: patrones de análisis, de diseño y de código, además, se describe al menos un ejemplo de los autores más representativos que corresponden a esta clasificación; en el cuarto y quinto punto se sugieren varias formas para seleccionar y utilizar el mejor patrón, respectivamente.

- La segunda parte es el desarrollo del caso de estudio: el ensamble de un jean. La dinámica seguida es como sigue: se describe el problema "el ensamble de un jean" en términos de los casos de uso de I. Jacobson, enseguida se señalan los sustantivos y verbos para identificar las clases y los objetos, posteriormente, se encuentran las relaciones entre clases y sus responsabilidades con las tarjetas CRC. Se obtiene un primer modelo del problema con el lenguaje unificado de modelación UML. Enseguida,

se implementa el modelo con el lenguaje de programación orientada a objetos Java.

- La tercera parte describe 7 patrónes de diseño que se utilizan para re-estructurar la implementación del caso de estudio de la segunda parte.

- La cuarta parte son las conclusiones, en donde se hace énfasis en los objetivos alcanzados.

Por último, se incluye la bibiografía con los autores y direcciones de www consultados, así como un glosario de términos con una explicación breve de las palabras o temas complementarios a los tratados.

En paralelo a la impresión de este trabajo en una tesis, se ha dispuesto en www en la siguiente dirección: http://uxmcc1.iimas.unam.mx/~jorge/patternDesign/SDPD.html.

Nota: La elaboración de éste trabajo tiene un valor estrictamente académico y no comercial. Aunque se trató en todo momento de ser original, muchas veces no fue posible, por lo que el uso sin autorización de citas, notas, diagramas, y formatos, que puedan violar los derechos reservados de un autor u libro, es intencionado, y su responsabilidad es única y exclusivamente del autor.

Ciudad Universitaria, 19 de enero de 1998.

Capítulo 1

Patrones

Capítulo 1. Patrones

Introducción

A mediados de los 80's nuevos lenguajes de programación dan impulso al modelo orientado a objetos. Una de ellas, la ingeniería de software orientada a patrones, persigue incorporar en el conocimiento del diseñador de aplicaciones aquella experiencia de los considerados como expertos en un dominio específico, la idea de registrar el conocimiento del diseño en una forma canónica se inicia en el campo de la arquitectura con Christopher Alexander. El es el primero en establecer la noción de patrón para la construcción de ciudades, pavimentación de calles, decoración de casas, etc. Durante más de 20 años, él y un grupo de investigadores del Centro para la Estructura del Medio Ambiente en Berkeley, California, han trabajado en el desarrollo de un sistema arquitectónico usando patrones. Han publicado diferentes libros [*16.17.18.19*] en donde plantean las bases teóricas para un nuevo enfoque en la arquitectura, la construcción y la planeación.

Los pioneros de los patrones en el desarrollo de software son Ward Cunnigham y Kent Bech [*44*]. Ambos se inspiraron en el trabajo de Alexander y adoptaron sus ideas aplicadas al desarrollo de software. Desarrollaron cinco patrones para el diseño de interfaces: Window per Task, Few Panes, Standard Panes, Nouns and Verbs y Short Menus, los cuales marcan el nacimiento de los patrones en la ingeniería de software.

El primer trabajo publicado de patrones es la tesis doctoral de E. Gamma en 1991 [*35*], escrita en alemán y que no obtuvo reconocimiento fuera de Europa Central. Gamma es reconocido por emplear, por primera vez, la metodología de análisis y diseño orientada a objetos usando patrones para resolver problemas de diseño en *frameworks* (ver *glosario*).

La rama de la ingeniería de software enfocada a patrones, realmente se inició con la publicación del libro Design Patterns [*3*], de E. Gamma, R. Helm, R. Johnson y J. Vlissides. El Catálogo de Patrones, nombre más común por el cual se le conoce, está dirigido a la etapa de diseño en el desarrollo de software. Gamma et al, retoman las ideas esenciales de Alexander, para expresar soluciones usando clases y objetos a problemas que no necesariamente tienen que ser específicos de un contexto, lo cual amplía la noción de patrón de Alexander.

Diferentes estilos, aplicaciones, formas y enfoques de patrones se han publicado, los más conosidos son los siguientes:

J. Coplien hace un análisis de la evolución de C++ [*34*] donde puntualiza que se caracteriza por la implementación de constructores de expresiones nativas de otros sistemas, para adquirir una funcionalidad que no le pertenece. El uso de los constructores es tan frecuente que se dice que se han *idiomatizado*, denominandolas ***idioms*** ó ***patrones de código***.

Duglas C. Schmidt [*45*] es uno de los más entusiastas de los patrones enfocados a sistemas distribuidos y redes. Inicialmente trabajó con el framework "Adaptive Communication Environment", el cual apoya la construcción de aplicaciones distribuidas. Ha publicado un gran número de patrones que son ampliamente utilizados en el desarrollo de comunicaciones.

Coad et al. [*21*] dirige su atención al análisis de sistemas orientados a objetos. Desarrollan patrones y

estrategias para la construcción del modelo de objetos. Una estrategia es la planeación de una acción para conseguir un objetivo. Un patrón es considerado como una plantilla de objetos con responsabilidades e interacciones, el cual puede ser aplicado una y otra vez por analogía.

Buchmann et al. [*20*] presentan un conjunto de patrones relacionados con diferentes grados de abstracción: de arquitectura, de diseño y de código (idioms). Los de arquitectura proveen el esqueleto de la aplicación, los de diseño complementan los de estructura y los de código hacen referencias a los existentes en C++ y Smalltalk. Desarrollan un criterio propio para clasificar patrones: interactivos, adaptables, de organización, de comunicación y de control, por lo que, afirman, los patrones forman un sistema.

Definición

*A*lexander [*16*] se refiere al término ***patrón*** de diferentes formas: *un patrón es una regla que expresa una relación entre un contexto, un problema y una solución; un patrón es una relación entre fuerzas que se dan en un contexto y una configuración que restringe las fuerzas; un patrón es una regla que explica como crear la configuración que resuelve las fuerzas antagónicas.*

Gramma el al, retoman la definición de Alexander y describen la que es mundialmente más popular: *un patrón es la solución a un problema recurrente en un contexto particular*, aplicable no sólo a la arquitectura sino también al diseño de software.

Coad et al., dan una definición tomada del diccionario: *un patrón es un modelo, una imitación, una plantilla de objetos con responsabilidades y relaciones.*

En este trabajo, el término patrón se utiliza en el siguiente sentido: ***es un modelo heurístico que representa el conocimiento abstracto de la solución a un problema recurrente en un contexto específico.***

Clasificación

*D*ada la gran diversidad de patrones existentes en todas las áreas de aplicación, es necesario establecer una clasificación que ayude al usuario final a escoger el mejor patrón que solucione de la mejor forma su problema. En esta parte del trabajo se promueve una clasificación apegada a los modelos de desarrollo de software: análisis, diseño e implementación. Se categorizan los patrones de tres autores principales Coad et al., Gamma et al, y Coplien, en patrones de análisis, de diseño y de código, respectivamente.

Patrones de Análisis

*U*n patrón de análisis es aquel que guía al usuario desde las fases iniciales del desarrollo de un proyecto hasta la identificación y elaboración del modelo de clases y objetos. Está dirigido a usuarios sin experiencia en la etapa del análisis y paso a paso los conduce en forma gradual e incremental a identificar los puntos críticos, áreas problemáticas, objetos, relaciones entre objetos, clases, y la elaboración del modelo de objetos.

De la gran variedad de patrones, los que mejor corresponden a esta etapa son los del libro de Coad et al., en el cual se desarrollan 148 estrategias y 31 patrones [*21*].

Estrategias. Son consideradas como consejos para conseguir objetivos, se clasifican en diversos grupos:

- **Actividades y componentes.** Guían los pasos para organizar actividades.
- **Identificación del propósito del sistema y sus características.** Describen el objetivo del sistema.
- **Selección de objetos.** Ayudan a identificar objetos.
- **Asignación de responsabilidades.** Facilitan la definición de las responsabilidades de las clases.
- **Determinación del dinamismo del sistema.** Diseño de diagramas de interacciones y escenarios.
- **Descubriendo nuevas estrategias y patrones.** Como encontrar nuevas estrategias y patrones.

Patrones. Son considerados como una plantilla de objetos relacionados, que pueden ser usados una y otra vez por analogía. Se clasifican en diversos grupos:

- **Fundamental.** Es la base de todos los demás.
- **De transacción.** Comprenden a los componentes para realizar transacciones.
- **De agregación.** Determinan los objetos que son contenedores.
- **De planeación.** Ayudan en las especificaciones de los planes de ejecución.
- **De relación.** Definen los tipos de relaciones entre los componentes.

El formato de las estrategias y patrones es un rectángulo con las siguientes características:

El de las estrategias se divide en dos: la parte superior identifica el número, nombre, y clasificación, la parte inferior describe sugerencias, recordatorios, ejemplos, etc. Por ejemplo, la estrategia # 25 "El modelo de componentes como una guía", ayuda a organizar los objetos que integraran el modelo, figura 1.1.

Los componentes que lo integran son 5:

- **PD (Dominio del problema).** Son las clases básicas relacionadas con el sistema.
- **HI (Interacción humana).** Son las ventanas y reportes.
- **DM (Administración de datos).** Aspectos relacionado con las bases de datos.
- **SI (Relaciones del sistema).** Relaciones con otros sistemas.
- **NT (No en este momento).** Conceptos que en este momento no se contemplan.

#25. "Guía para el modelo de componentes", selección de objetos(modelo de componentes)
• Identificar los objetos usando el modelo de componentes • PD: (dominio del problema) • DM: administración de datos • HI: interacción humana • SI: interacción con otros sistemas • NT: no en este momento

Figura 1.1. Estrategia #25 "El modelo de componentes como una guía".

El rectángulo de presentación de los patrones se divide en tres: la primera parte registra el número, nombre, y clasificación; la segunda parte tiene el diagrama de clases y la tercera parte describe los componentes, relaciones, ejemplos y notas. Por ejemplo, el patrón #1 "Conjunto-Trabajador", captura la relación uno-muchos entre dos clases. Se considera como un patrón principal del cual se derivan todos los demás patrones, figura 1.2.

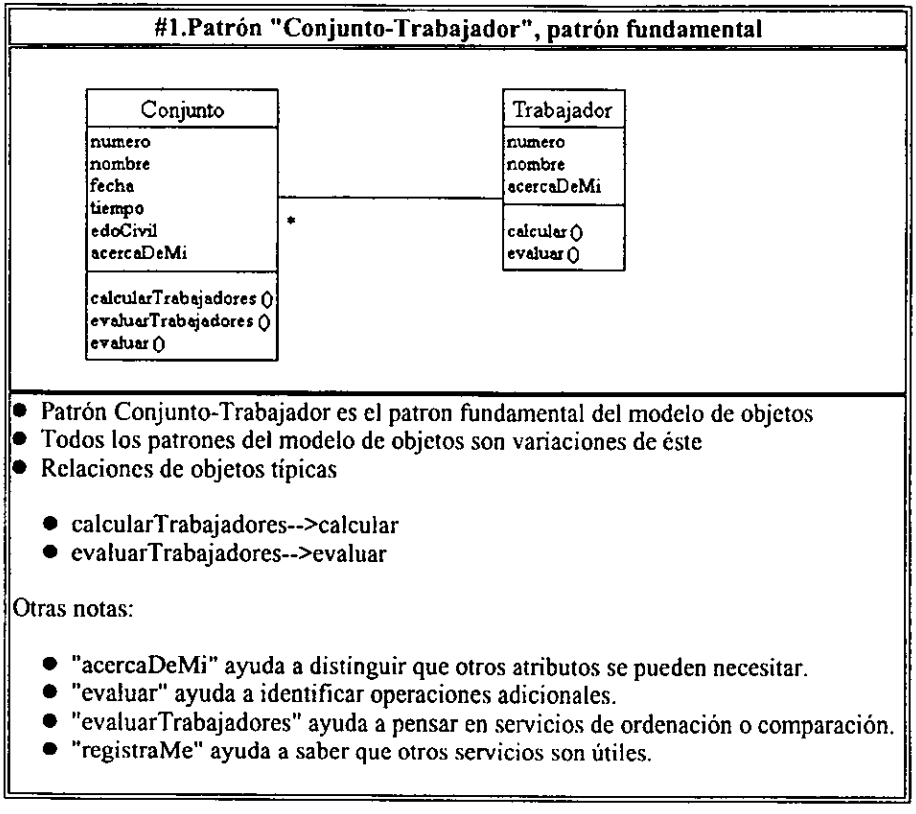

Figura 1.2. Patrón #1 "Conjunto-trabajador".

La finalidad de las estrategias y patrones de análisis es reducir el tiempo que le lleva a un usuario llegar a ser un experto en la construcción del modelo de objetos.

Patrones de Diseño

Un patrón de diseño describe soluciones a problemas específicos en el diseño de software orientado a objetos. Capturan la experiencia de diseñadores expertos en la solución de problemas recurrentes. Representan diseños utilizados en sistemas complejos y a gran escala.

Cada patrón de diseño nombra, explica, analiza, evalua e implementa el diseño de una solución recurrente, que ha demostrado ser eficiente en cuando menos dos o más sistemas orientados a objetos. El

uso adecuado del patrón es una garantía para que el diseño sea sencillo, elegante, flexible y reusable.

Los patrones de diseño de Gamma et al., son los más representativos de esta clasificación. "The Gang of Four", sobrenombre por el cual se les conoce, escriben el libro "*Pattern Design*", que comprende 24 patrones de diseño de diferentes áreas organizados en un catálogo [*3*]. A diferencia de otros patrones, como los presentados en el libro "*A System of Patterns*" de Buschmann et al, [*20*] que están integrados en un sistema, los de Gamma et al. son diseños dispersos organizados con base a dos criterios: *alcance (scope)* y *propósito*.

Alcance (Scope). Se refiere al campo de aplicación, el cual puede ser estático (class) o dinámico (object):

- **Estático.** Se refiere a las relaciones entre las clases y las subclases , las cuales se establecen por mecanismos de herencia y son estáticas, definidas en el momento de compilación.
- **Dinámico.** Se refieren a las relaciones entre los objetos, las cuales pueden cambiar en tiempo de ejecución y por tanto son más dinámicas.

Propósito. Discreciona la funcionalidad, comprende tres subdivisiones: de creación, de estructura y de desempeño.

- **De creación.** Describen la forma de crear objetos.
- **De estructura.** Describen la estructura y composición de clases y objetos.
- **De desempeño.** Caracterizan la forma como se comunican las clases y los objetos.

En la figura 1.3 se observa que la mayoría de los patrones corresponden a la clasificación de alcance dinámico.

		Propósito		
		De creación	De estructura	De desempeño
Alcance	Estático	Método de fabricación	Adaptador(class)	Interpretador Método Template
	Dinámico	Fábrica Abstracta Constructor Prototipo Individual	Adaptador Separador Compositor Decorador Fachada Fragmentador Sustituto	Cadena de Responsabilidades Comando Iterador Mediador Temporal Observador Estado Estrategia Validador

Figura 1.3 . Clasificación de los Patrones de Diseño.

Nota: El nombre original de los patrones ha sufrido diversos cambios desde que Gamma los publicó en Alemán, los nombres de la tabla 1.3 no son sólo traducciones literales, sino también reflejan el objetivo y contexto de la aplicación. La figura 1.4 muestra el nombre propuesto en español de su correspondiente en ingles.

Método de Fabricación	Factory Method
Fábrica Abstracta	Abstract Factory
Constructor	Builder
Prototipo	Prototype
Individual	Singleton
Adaptador	Adapter
Separador	Bridge
Compositor	Composite
Decorador	Decorator
Fachada	Facade
Fragmentador	Flyweight
Substituto	Proxy
Interpretador	Interpreter
Método de Plantilla	Template Method
Cadena de Responsabilidades	Chain of Responsability
Comando	Command
Iterador	Iterator
Mediador	Mediator
Temporal	Memento
Observador	Observer
Estado	State
Estrategia	Strategy
Visitador	Visitor

Figura 1.4. Relación del nombre de los patrones en ingles y español.

Mientras que el objetivo de un patrón de análisis, es conducir al usuario paso a paso hasta la elaboración del modelo de objetos, los de diseño tienen varios objetivos:

- **Identificar objetos.** Aunque la identificación de objetos, por lo general, se realiza en la etapa del análisis, existen otros que se determinan en tiempo de ejecución, por ejemplo: "Compositor" compone objetos que se accesan uniformemente.
- **Granularidad de los objetos.** El tamaño y número de los objetos es variable y depende de la aplicación, por ejemplo: "Fachada" representa subsistemas completos como objetos; "Fragmentador" describe las diferentes representaciones de un objeto; "Método de Fabricación" crea objetos individuales; "Constructor" crea familias de objetos; "Visitador" agrega funcionalidad a un objeto y "Commando" distribuye responsabilidades.
- **Extensión de la interfaz.** La interfaz es el medio por el cual un objeto se da a conocer y se comunica con otro objeto. Los patrones de diseño ayudan a definir la interfaz correcta, por ejemplo: "Temporal" describe como encapsular y guardar el estado interno de un objeto, para que

posteriormente pueda regresar a su estado anterior; "Substituto" difiere su control a otro objeto; "Compositor" define objetos compuestos y "Decorador" añade responsabilidades dinámicamente a un objeto.

- **Programar para la interfaz y no la implementación.** Se debe favorecer la creación de clases generales (clases abstractas) de donde se desprendan las demás clases y no únicamente clases específicas y/o concretas. Entre más clases abstractas se tengan, más dinámico es el diseño. En general los patrones clasificados "de creación" aseguran que el sistema esté escrito en términos de clases abstractas.
- **Favorecer la composición de objetos en vez de la herencia.** La herencia de clases concretas muestrala implementación de la clase madre a la hija, lo que crea una dependencia y rompe el encapsulamiento [*46*], la composición de objetos:
  - Se define en tiempo de ejecución, por lo que fuerza a especificar muy bien las interfaces de las clases.
  - El comportamiento del sistema se basa en la composición de los objetos.
  - Se promueve el ensamble de objetos, más que la creación individual.
  - El sistema se mantiene manejable: pocas clases y muchos objetos.
  - No se rompe el encapsulamiento.
  - Las clases se enfocan a tareas concretas.
- **Delegación.** Un objeto delega sus responsabilidades a otro, cuando el primero mantiene una referencia del segundo. La referencia se puede cambiar en tiempo de ejecución, por lo que la delegación es un caso particular de la composición de objetos, varios patrones de diseño utilizan este concepto: "Estado" define un objeto que cambia su comportamiento cuando su estado interno cambia, el objeto parece que cambia de clase, "Estrategia" define un objeto que puede implementar diferentes algoritmos, cada uno es una diferente estrategia, "Mediador" especifica un objeto que sirve de intermediario entre otros objetos, "Cadena de Responsabilidades" define una cadena de objetos dando la alternativa a cada uno de responder a un mensaje, "Separador" separa la implementación de su interfaz, permitiendo que las dos varien independientemente.
- **Reusabilidad.** La maximización de la reusabilidad radica en la anticipación a nuevos requerimientos y de cambios a los existentes; el sistema debe ser lo más flexible para que cambie dinámicamente. Los patrones de diseño dejan que algunos aspectos del sistema varien independientemente unos de otros; algunas causas de rediseño, junto con los patrones que pueden ayudar, son los siguientes:
  - **Creación de un objeto especificando su clase explícitamente.** Especificando el nombre de la clase cuando se crea el objeto, se liga el objeto a una implementación en particular, en lugar de a una interfaz. Los objetos se deben de crear indirectamente: "Fábrica Abstracta", "Método de Fabricación" y "Prototipo". En la parte tres de este trabajo se explican ampliamente los dos primeros patrones, Prototipo define una interfaz comuún a un conjunto de objetos concretos, cada objeto personaliza su interfaz, la creación de los objetos sucede cuando se instancia un objeto concreto teniendo como tipo del objeto a Prototipo.
  - **Dependencia de una operación.** Cuando se especifica una operación en particular, únicamente se obtiene una respuesta. Se debe dar oportunidad a otros objetos de responder a una solicitud: "Cadena de Responsabilidades" y "Comando". Ambos patrones permiten compartir la responsabilidad que tienen con otros objetos.
  - **Dependencia de la plataforma (harware o software).** El diseño del sistema no debe depender de la plataforma: "Fábrica Abstracta" y "Separador" . Ambos patrones definen diversas formas para que un misma implementación sea independiente de la arquitectura.
  - **Dependencia en la representación o implementación.** El cliente no debe saber como los objetos se representan, almacenan, localizan e implementan: "Fábrica Abstracta",

"Separador" , "Temporal" y "Substituto".

- **Dependencia de los algoritmos.** Los algoritmos que probablemente cambien se deben aislar: "Constructor", "Iterador", "Estrategia", "Método de Plantilla", y "Visitador".
- **Debil acoplamiento (Tight coupling).** Un sistema fuertemente acoplado es monolítico, se debe promover un acoplamiento debil y en diferentes niveles: "Fábrica Abstracta", "Separador", "Cadena de Responsabilidades", "Comando", "Mediador" y "Observador".
- **Extendiendo la funcionalidad con subclases.** Se debe balancear el diseño estático (herencia) con el dinámico (composición y delegación): "Separador", "Cadena de Responsabilidades", "Compositor" , "Decorador", "Observador", y "Estrategia".
- **Inhabilidad para alterar una clase convenientemente.** Cuando se requiere modificar una clase, puede ser que no se tenga el código fuente, o que se tengan que cambiar varias clases: "Adaptador", "Decorador" y "Visitador".

El formato de los patrones de diseño es el siguiente:

- **Nombre y clasificación.** El nombre distingue un patrón de otro; la clasificación determina el propósito y si es de alcance estático o dinámico.
- **Objetivo (Intent).** Objetivo del patrón.
- **También conocido como.** Otros nombres por los cuales también se le conoce.
- **Motivación.** Cual es el problema de diseño que originó la solución que ofrece el patrón.
- **Aplicación.** cuando se debe usar el patrón.
- **Estructura.** Representación gráfica de las clases y objetos del diseño del patrón.
- **Participantes.** Componentes que participan, clases y objetos.
- **Colaboraciones.** Como cooperan los componentes.
- **Consecuencias.** Ventajas y desventajas al usar el patrón.
- **Implementación.** Aspectos que se tienen que considerar cuando se implementa el patrón.
- **Código de un ejemplo.** Extractos de código de alguna aplicación.
- **Sistemas en los cuales se ha usado.** Sistemas conocidos en los cuales se ha usado el patrón, se mencionan por lo menos dos.
- **Patrones relacionados.** Con que otros patrones, el actual se puede utilizar.

En la siguiente parte, "Patrones de diseño implementados", se desarrollan 7 patrones de diseño que complementan el caso de estudio: "el ensamble de un jean".

Patrones de Código

El término "idiom" o patrón de código, es una expresión o construcción especifica del lenguaje natural y de los lenguajes de programación. En los lenguajes de programación se utilizan para definir conjuntos de proposiciones complejas que dan la apariencia de formar parte del lenguaje, aunque en realidad no lo sean. El significado de un "idiom" no puede ser derivado de los elementos que lo integran, determina un funcionalidad propia del lenguaje que no está definida y que es transparente al usuario.

Aunque todos los lenguajes de programación hacen uso extenso de los "idioms", en esta sección se tratará únicamente su vinculación con el diseño de C++, para ello, se tomó como base el excelente tratado de programación de C++ por Coplien: "*Advanced C++: programming styles and idioms*" [*34*].

La evolución de C++ ha seguido dos sentidos:

- La adopción de conceptos metodológicos del modelo orientado a objetos, desarrollados en las universidades y centros de investigación.
- La creación de expresiones y constructores que le permitan incorporar características de otros lenguajes.

El desarrollador de C++, Stroustrup [*52*], decidió que las tareas que dependen del tiempo de compilación, como la conversión de tipos, formarán parte del lenguaje, mientras que otras que se definen dinámicamente, como la administración de memoria, utilizarán expresiones y constructores como conceptos más abstractos para llamar las definidas en otros sistemas, por ejemplo el sistema operativo. El uso de expresiones y constructores se volvió tan frecuente que se adoptaron como parte de la gramática del lenguaje, es decir, se "idiomatizaron", de donde viene la palabra "idiom".

Antes de profundizar más en el tema, el siguiente ejemplo aclara un poco su uso y significado:

- Los operadores de entrada << y de salida >>, no son parte del lenguaje "C++", ni de su antecesor "C con clases". La complejidad no radica en el compilador, ni en la aplicación, pero si en una biblioteca que se accesa con la directiva #include y que sobrecarga los operadores mencionados, dando la apariencia de que los operadores de cambio de bits "<< y >>", fueron originalmente diseñados para ser operadores de entrada y salida.

Las áreas que C++ no define, son precisamente las que le dan más flexibilidad respecto a otros lenguajes con modelos de programación más completos. Las características inherentes de C++ es que es un lenguaje de bajo nivel, que con el empleo en conjunto de "idioms", constructores y sobrecarga de operadores le dan la jerarquía de uno de alto nivel.

Uno de los "idioms" más importantes es la forma canónica de una clase. El siguiente ejemplo muestra como crear variables de clases que pueden ser asignadas, declaradas y pasadas como argumentos siguiendo la sintaxis de C.

Por convensión, la interfaz de la clase String debe seguir un formato para ocultar los detalles de la representación de cadenas en C. Si se sigue éste formato para describir las clases, las variables (objetos) creadas de las clases pueden ser asignadas, declaradas, y pasadas como argumentos en la misma forma como si se tratarán de variables de C.

```
class String {
public:
friend String operator+(const char*, const String&);
String operator+(const String&) const;
int length() const;
.... String();
String(const String&);
String& operator=(const String&);
~String();
String(const char *);

private:
char *rep;
```

```
};
```

La declaración de cada método es el siguiente:

- Constructor por omisión:

```
String::String() {
rep = new char[1];
rep[0]='\0';
}
```

Por ejemplo, éste constructor inicializa cada uno de los elementos del siguiente arreglo: String myArray[10].

- Constructor que crea una copia de una cadena existente:

```
String::String(const String& s) {
rep = new char[s.length() + 1];
::strcpy(rep,s.rep);
}
```

La copia es lógica y no física.

- Operador de asignación = :

```
String& String::operator=(const String s) {
if(rep != s.rep) {
delete[] rep;
int lengthOfOriginal = s.length() + 1;
rep = new char[lengthOfOriginal];
::strcpy(rep, s.rep);
}
return *this;

}
```

La asignación actua en forma similar al constructor anterior, excepto que regresa la dirección del objeto copiado. Notese que se esta usando el operador de asignación para hacer una copia de un string, esto es, se esta sobrecargando el operador de asignación. Por ejemplo, String x,y; asigna y a x.

- Destructor:

```
String::~String() {
delete[] rep;
}
```

- Constructor que crea una cadena de una definida en C:

```
String::String(const char *s) {
int lengthOfOriginal = ::strleng(s) + 1;
rep = new char[lengthOfOriginal];
::strcpy(rep.s);
}
```

Aparte de ser un constructor, define la forma en que se deben crear las cadenas. Por ejemplo, int hash("literal") llama automáticamente éste constructor y crea la cadena literal.

- Longitud de una cadena:

```
int String::length() const {
return::strlen(rep);
}
```

El ejemplo anterior, describe un patron o "idiom", que se debe seguir cuando se implemente cualquier clase. Siguiendo el formato, cualquier clase W esta caracterizada por lo siguiente:

- Un constructor por omisión: W::W()
- Un constructor que crea una copia de una cadena existente: W::W(**const** W&)
- Un operador de asignación: W& operator=(**const** W&)
- Un destructor: W::~W()

Cuando usar el patrón de código o "idiom". En general, se debe de usar la forma canónica si:

- Se desea asignar los objetos de la clase, o se quieren pasar esos objetos como parámetros por valor a una función.
- El objeto contiene apuntadores de otros objetos, o el método constructor realiza un delete() sobre los atributos del objeto.

La clase String es considerada como una forma canónica ortodoxa, y es la base para el resto de los "idioms" desarrollados por Coplien en [*34*].

Cómo seleccionar un patrón?

*U*no de los problemas más frecuentes cuando se está trabajando con patrones, es encontrar la abstracción de la solución encapsulada en el modelo heurístico (clases y objetos) y aplicarla usando la analogía en el dominio y contexto específico del usuario. La clasificación anterior (patrones de análisis, de diseño y de código) discrimina los patrones en las fases que corresponden al desarrollo de software (análisis, diseño e implementación). Los patrones que corresponden a cada una de estas fases son bastante numerosos, además de ser similares y complejos, por lo que no es fácil saber cual escoger y algunas veces la solución que "dicen" ofrecer no es clara.

Aunque no hay reglas específicas para seleccionar un patrón, son útiles las siguientes sugerencias:

- Identificar en que fase del desarrollo se esta:

- Si es de análisis:
  - Identificar el problema respecto a la clasificación de las estrategias y patrones. Si es una tarea que se desea realizar se selecciona una estrategia, u organizar un conjunto de objetos se selecciona un patrón.
  - Analizar que se desea hacer y las estrategias y patrones que pueden ayudar, por ejemplo: seleccionar objetos (estrategias 13-24), componentes del modelo (estrategias 25-33), objetos que integran el modelo (patrones de agregación), etc.
  - Entender la interacción entre las entrategias y los patrones; por lo general no sólo una estrategia y/o patrón es el adecuado (a), sucede frecuentemente que una estrategia y/o patrón es la generalización de otro, o bien dos o más realizan la misma tarea pero con diferentes sugerencias.
- Si es de diseño:
  - Identificar el problema respecto a la clasificación de los patrones de diseño: propósito y alcance (scope).
  - Analizar el enfoque que se le desea dar al problema: estático o dinámico.
  - Estudiar la interacción entre los patrones candidatos. Como se relacionan, cuando uno sustituye a otro, que cambios necesita uno para desempeñar la tarea de otro, etc.
  - Evaluar las ventajas y desventajas del nuevo diseño, integrantes, colaboradores, etc.
- Si es de código:
  - La dependencia que existe entre un patrón de código y el lenguaje de programación hacen difícil sugerir consejos prácticos que no sean otros que se puedan consultar en los manuales del lenguaje. Citando a Coplien, se tiene que identificar el problema con la estructura en el lenguaje, por ejemplo, si se desea crear una cadena, referirse a la forma canonica de una clase, si se desea implementar un recolector de basura referirse a la implementación del algoritmo de Baker [*47*] o bien estudiar el código de la implementación de figuras geométricas de Coplien.

Cómo usar un patrón?

Pueden seguirse varios enfoques, el siguiente es uno de ellos:

- Leer el patrón haciendo énfasis en el objetivo, motivación y consecuencias.
- Estudiar la estructura, participantes y colaboradores.
- Estudiar los extractos de código (patrones de diseño y de código).
- Analizar los patrones relacionados.
- Implementar el patrón haciendo los ajustes necesarios.

Capítulo 2

Caso de estudio: ensamble de un jean

Capítulo 2. Caso de Estudio

Objetivo

*E*jemplificar el proceso textil del "ensamble de un jean" utilizando la metodología de análisis, diseño e implementación orientada a objetos, con los siguientes enfoques:

- Análisis. Modelo de los casos de uso de I. Jacobson [*10, 14*].
- Diseño. Patrones de Diseño de Gamma et al. [*3*].
- Implementación. Lenguaje Java.

Análisis

*E*l modelo de los casos de uso consiste, principalmente, de tres conceptos:

- El sistema bajo estudio, figura 2.1.
- El sistema y su medio ambiente, figura 2.2.
- Un actor y la interacción con el sistema "caso de uso", figura 2.3.

El sistema bajo estudio se denomina "The business system", comprende dos vistas, la externa y la interna. La externa describe el medio ambiente del sistema, especificamente, los agentes que interactuan con él. La interna se refiere a su comportamiento interno, los elementos que lo integran así como sus interacciones, figura 2.1.

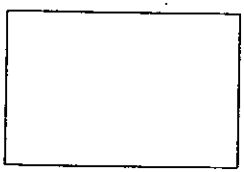

Figura 2.1. Sistema Textil

Analizando la vista externa, las interacciones que realiza cada agente respecto al sistema se denominan *roles*. El agente, entonces, juega un rol en el sistema. El agente, es el ente que representa el (los) rol (es), por lo que se le denomina *actor*. Un actor, por tanto, juega uno o varios roles respecto al sistema. En la figura 2.2 se observa el actor representado por la figura de un muñeco y el sistema bajo estudio por un recuadro.

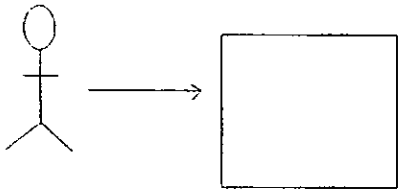

Figura 2.2. Sistema Textil y su Medio Ambiente

En este sistema, únicamente se diseñó un actor, con la finalidad de limitar el esquema de estudio. El actor es el cliente que acude al sistema y demanda un servicio. Un segundo actor podría ser el proveedor de la tela, o de los hilos, etc. La acción que demanda el actor del sistema corresponde a un caso de uso. En nuestro ejemplo, también limitamos la interacción del cliente a un sólo caso de uso "coser un jean". Un *caso de uso* es una secuencia de pasos (transacción) que aporta un valor medible al actor, ayudan a identificar las partes internas, externas, responsabilidades e interacciones del sistema [*10,14*]. El símbolo para denotar un caso de uso es un círculo, figura 2.3. Los casos de uso forman parte del sistema y corresponden a su vista interna.

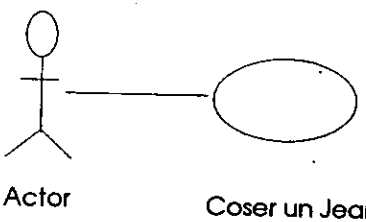

Figura 2.3. Caso de Uso: Coser un Jean

Los casos de uso representan procesos dentro del sistema, se les suele nombrar con un verbo o un verbo seguido de un sustantivo. Si el nombre del caso de uso no aporta información sobre su objetivo, conviene describirlo. En el caso de estudio que se esta analizando, la descripción es la siguiente:

Coser un jean: Un cliente , Sr. Melquiades, entra a la fábrica de jeans "Textiles Azul" y solicita el catálogo de jeans. La empleada, Srita. Matilde, le muestra el catálogo de la colección de invierno. Despues de hojear y hacer diversas preguntas sobre el estilo, forma, textura y precio por volumen, el Sr. Melquiades decide hacer un pedido de jeans estilo clásico. La Srita. Matilde solicita al departamento de inventarios las existencias del jean estilo clásico. Despues de certificar que no hay existencias en almacén, levanta un pedido con fecha de entrega de mercancía de no más de dos meses, ya que de lo contrario el producto estaría fuera de temporada. Enseguida, la Srita. Matilde, envía la solicitud al departamento de planeación y control de producción. El encargado de planeación, Sr. Cleofas, revisa las existencias en inventario y en las áreas de máquinas (jeans actualmente produciendose). Como no hay existencias, don Cleofas, hace una orden de varios metros de tela de mesclilla azul al almacén, etc.

El siguiente paso es cortar las piezas del jean de las telas extendidas. De un jean clásico, figuras 2.4 y 2.5, se obtienen los moldes por pieza que se van a dibujar en la tela para que después se corten.

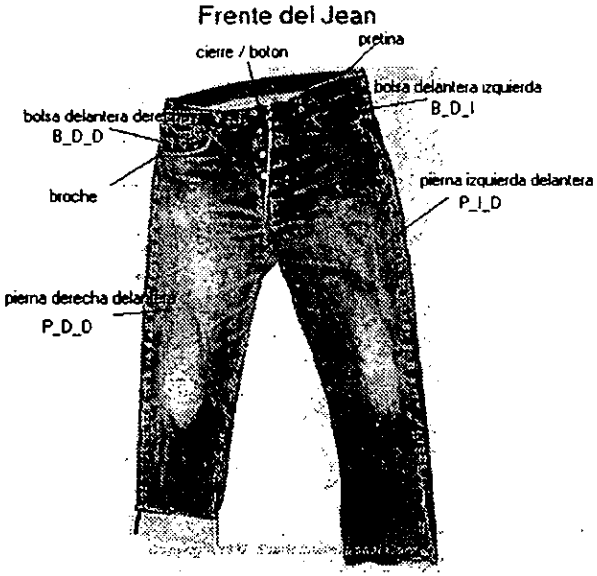

Figura 2.4 . Frente del Jean

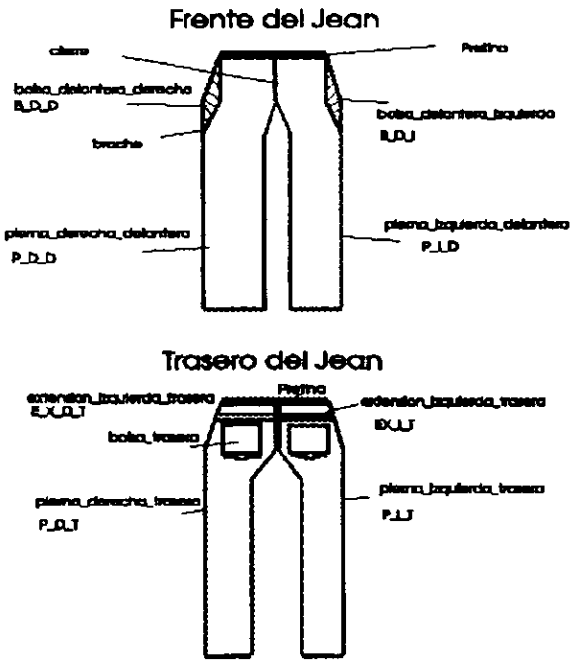

Figura 2.5 Descomposición del Jean

La jerarquía de las piezas se observa en la gráfica de Gantt de la figura 2.6. Una vez cortadas las piezas, éstas se distribuyen en las áreas de las máquinas para que inicie el proceso de producción.

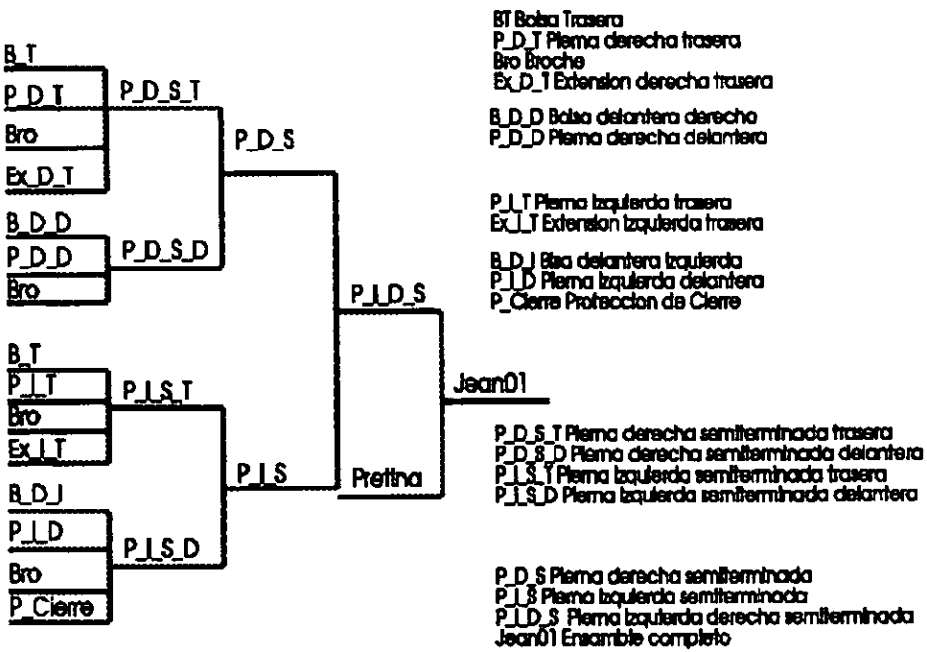

Figura 2.6 Diagrama de Gantt. Ensamble de un Jean.

El proceso de producción tiene cierto orden. Los ensambles no se pueden coser si previamente no se tienen los elementos que los integran. Así, por ejemplo, para ensamblar Jean01, se necesita la disponibilidad de P_I_D_S, y Pretina; P_I_D_S necesita de P_D_S y P_I_S, etc.

El proceso de producción sigue la siguiente dinámica: los elementos que constituyen cada ensamble deben de estar disponibles antes de formar parte del ensamble, así ***B_T***, ***P_D_T***, ***Bro*** y ***Ex_D_T*** son *cosidos* para formar ***P_D_S_T***; ***P_D_S_T*** y ***P_D_S_D*** también son *cosidos* para formar parte de ***P_D_S***, etc.

Limitando el caso de uso a la descripción del proceso de producción, la interacción entre los objetos se describe en un diagrama de secuencias (ver *glosario*); en la figura 2.7 se observa el diagrama de secuencias del caso de uso "coser un jean", el cual por razones de espacio únicamente muestra la comunicación entre 7 objetos.

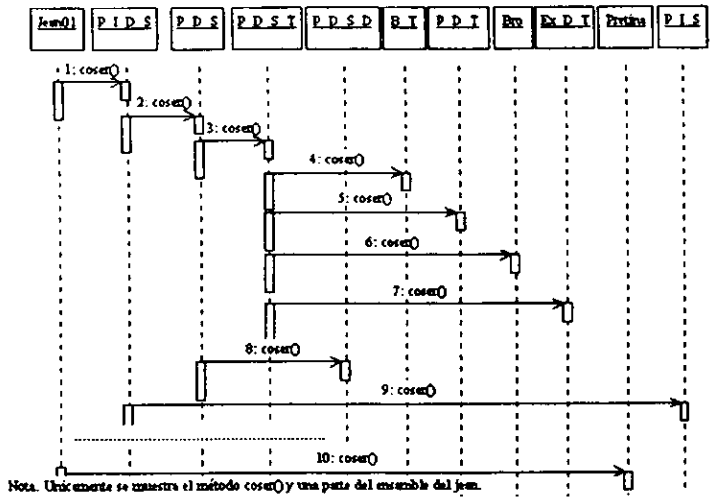

Figura 2.7 . Diagrama de Secuencias del caso de de uso Cocer un Jean.

El siguiente paso es encontrar las responsabilidades de las clases de los objetos que integran el jean. Para ello, se emplea la metodología de las tarjetas CRC, ver *glosario*. La metodología consiste en anotar en tarjetas de 10 x 15 cm, el nombre de la clase, las responsabilidades y los colaboradores. El caso de Jean01, P_I_D_S y P_D_S con la responsabilidad coser, se muestra enseguida:

Jean01	
Responsabilidades	**Colaboradores**
coser	P_I_D_S, Pretina

P_I_D_S	
Responsabilidades	**Colaboradores**
coser	P_D_S, P_I_S

P_D_S	
Responsabilidades	**Colaboradores**
coser	P_D_S_T, P_D_S_D

Aunque en la literatura del análisis, diseño e implementación orientada a objetos, no hay una clara separación entre el análisis y el diseño, se puede afirmar que una vez identificados los objetos, la comunicación entre los objetos, las clases y las responsabilidades de las clases que comprenden el dominio del problema, el proceso de retroalimentación del análisis en su fase inicial ha concluido. Booch [9] lo plantea de la siguiente forma: el análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

Diseño

Apoyados en la gráfica de Gantt del Jean y en el análisis previo, se obtuvo el diseño de la figura 2.8 siguiendo la notación de UML (Unified Method Languaje), ver *apèndice*

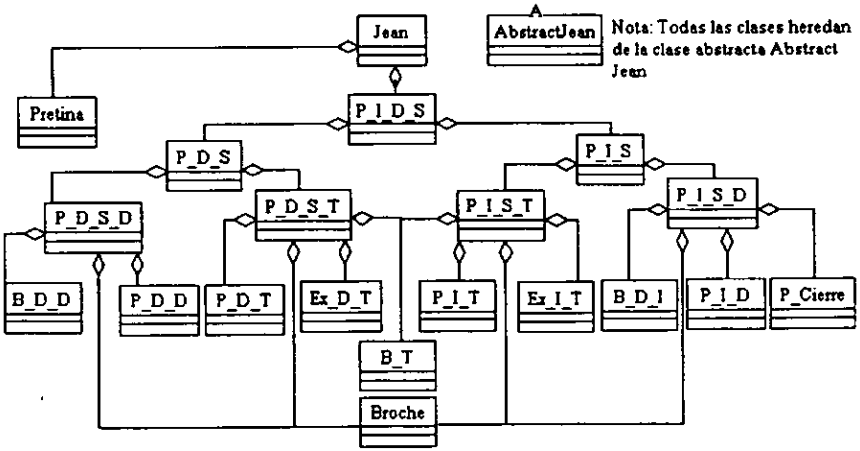

Figura 2.8 . Diagrama de clases del ensamble de un jean.

En él, se observa lo siguiente:

- Los objetos de las clases que son ensambles: Jean, P_I_D_S, P_D_S,... P_I_S_D, representan el hecho de que se integran de otros objetos, utilizando el simbolo de agregación <>.
- La clase AbstractJean es una clase abstracta que capta los atributos y métodos comúnes a todas las clases. Las subclases heredan e implementan las operaciones abstractas.
- Los objetos que pertenecen a varios ensambles son identificados, por ejemplo: B_T y Broche.
- Se esta favoreciendo la composición de objetos, por lo que, el desempeño del modelo depende más de la comunicación de sus objetos integrantes, que de las clases mismas. Bajo este enfoque, se reducen las clases y se aumentan los objetos. El modelo es, por tanto, más dinámico.

Implementación

La etapa final del ciclo del desarrollo de software es la implementación. En esta sección, se desarrollan los programas para el diseño previo en el lenguaje Java. La implementación lograda es la más sencilla posible, a fin de mantener la esencia del proceso de producción del jean. Se explican los aspectos principales de las clases con extractos de código. Alternativamente, el programa completo se encuentra disponible en linea: *Vnaive.java*.

La clase abstracta AbstractJean, tiene una variable booleana "cosido", que indica si el objeto ya se cosió, y de un método abstracto "coser" que cada subclase implementa.

AbstractJean

```
abstract class AbstractJean
{
protected boolean cosido = false;
...
public abstract void coser();
}
```

Ensambles

Jean

La clase Jean tiene como atributos dos variables del tipo P_I_D_S y Pretina respectivamente. Cuando es invocado el método coser() de Jean, jean.coser(), cada objeto p_i_d_s y pretina invoca su propio método coser. Si estos objetos son compuestos, es decir, estan formados de otros objetos, cada objeto integrante invoca su método coser(). Los ensambles son objetos compuestos, las piezas no.

```
class Jean extends AbstractJean
{
private P_I_D_S p_i_d_s;
private Pretina pretina;
....
public void coser()
{
p_i_d_s.coser();
pretina.coser();
cosido=true;
System.out.println(miNombreEs() + "-----> ensamblado");
}
}
```

P_I_D_S

```
class P_I_D_S extends AbstractJean
{
private P_D_S p_d_s;
private P_I_S p_i_s;
...
public void coser()
{
p_d_s.coser();
p_i_s.coser();
cosido=true;
System.out.println(miNombreEs() + "-----> ensamblado");
}
```

}

Piezas

Pretina

```
class Pretina extends AbstractJean
{
public Pretina(...){...} // constructora
....
public void coser()
{
System.out.println(miNombreEs() + -----> cosido");
cosido = true;
}
}
```

P_Cierre

```
class P_Cierre extends AbstractJean
{
public P_cierre(..){..} // constructora
...
public void coser()
{
System.out.println(miNombreEs() + -----> cosido");
cosido = true;
}
}
```

Los objetos que no son ensambles son objetos primitivos; los únicos atributos que tienen son aquellos que heredan.

Las subclases de AbstractJean implementan el método abstracto coser(). En un objeto compuesto, sus integrantes invocan respectivamente su método coser(). Por ejemplo, P_I_D_S implementa el método coser() dejando que los objetos que lo integran invoquen a su vez su método coser().

P_I_D_S

```
class P_I_D_S extends AbstractJean
{
public void coser()
{
p_d_s.coser(); //p_d_s es tambien un ensamble
p_i_s.coser(); //p_i_s es tambien un ensamble
cosido = true; //indicación de que se esta cosiendo
System.out.println(miNombreEs()+"...---> ensamblado");
```

```
/* El método miNombreES() imprime el nombre de cada objeto.

}
}
```

La clase P_D_S implementa el método coser() en forma semejante, sus elementos (objetos) p_d_s_t y p_d_s_d invocan respectivamente su método coser(). La recursividad se da hasta que el objeto en cuestión sea un objeto primitivo.

La clase P_D_S_T tiene 4 atributos: b_t, p_d_t, broche y ex_d_t.

El método coser() de P_D_S_T tiene la siguiente forma:

P_D_S_T

```
class P_D_S_T extends AbstractJean
{
.....
public void coser()
{
bt.coser();
p_d_t.coser();
broche.coser();
ex_d_t.coser();
cosido = true;
System.out.println(miNombreEs()+"--->ensamblado");
}
}
```

Los objetos que son parte de P_D_S_T son objetos primitivos, no tienen descendencia, lo que se observa en la gráfica de Gantt, figura 2.6.

El método coser() de los objetos primitivos es mucho más sencillo, únicamente indican cuando ya fueron cosidos.

P_D_T

```
class P_D_T extends AbstractJean
{
.....
public void coser()
{
coser = true;
System.out.println(miNombreEs()+"---> cosido");
}
}
```

En forma semejante, los objetos primitivos implementan el mismo método; notese que el primer método coser() en ejecutarse es el de los objetos primitivos, enseguida, el control se regresa al método que los llamó, que pertenece a un ensamble. Un ensamble se dice que ya se ensambló si los elementos que lo forman ya ejecutaron su método coser().

La clase Vnaive contiene el método main() y la creación de los objetos.

Vnaive

```
class Vnaive
{
public static void main(String args[])
throws java.io.IOException
{
// ******* Piezas: Objetos Primitivos **********
Pretina pretina = new Pretina("PRETINA");
B_T b_t1 = new B_T("B_T1");
P_D_T p_d_t = new P_D_T("P_D_T");
Broche broche1 = new Broche("Broche1");
....
//***** Ensambles: Objetos Compuestos ******
P_D_S_T p_d_s_t = new P_D_S_T("P_D_S_T", b_t1, p_d_t, broche1, ex_d_t);
P_D_S_D p_d_s_d = new P_D_S_D("P_D_S_D", b_d_d, p_d_d, broche2);
P_I_S_T p_i_s_t = new P_I_S_T("P_I_S_T", b_t2, p_i_t, broche3, ex_i_t);
P_I_S_D p_i_s_d = new P_I_S_D("P_I_S_D", b_d_i, p_i_d, broche4, p_cierre);
.....
Jean Jean01 = new Jean("Jean01", p_i_d_s, pretina);
// ********Simulacion de Produccion**********
Jean01.coser();
}
```

Por último, en el método main() de la clase Vnaive, se crean los objetos invocando a sus respectivos constructores con una cadena que indica su nombre. En el caso de la creación de los objetos compuestos, es necesario añadir además del nombre, los objetos que lo forman.

Por ejemplo, para que el compilador no marque error al crear un objeto del tipo P_D_S_T:

```
P_D_S_T p_d_s_t = new P_D_S_T("P_D_S_T", b_t1, p_d_t, broche1, ex_d_t);
```

Es necesario crear primero los objetos que lo forman:

```
B_T b_t1 = new B_T("B_T1");
P_D_T p_d_t = new P_D_T("P_D_T");
Broche broche1 = new Broche("BROCHE");
Ex_D_T ex_d_t = new Ex_D_T("Ex_D_T");
....
```

En el diseño de la figura 2.8 y en la gráfica de Gantt, figura 2.6, se observa que la clase B_T es común a

P_D_S_T y P_I_S_T. La clase Broche es común a P_D_S_T. P_D_S_D. P_I_S_T y P_I_S_D. Sin embargo, en el código del programa completo se necesitan crear 2 objetos de B_T, y 4 de Broche.

Por último, una vez que se han creado todos los objetos, tanto primitivos como compuestos, solo se tiene una invocación del método coser().

```
Jean01.coser();
```

La invocación genera una reacción en cadena; cada objeto que forma parte de Jean01 manda un mensaje a su método coser(). Si este, a su vez, está formado de otros, cada uno de ellos también invoca su propio método coser(). Los mensajes en cadena siguen hasta llegar a los objetos primitivos. Enseguida viene el regreso de la invocación y solo es hasta este momento cuando realmente el objeto compuesto cambia de estado, es decir, cosido = true, o lo que es equivalente, se ensambla. El método coser() es un ejemplo de método polimórfico por inclusión [*15*].

La implementación que se ha logrado, presenta los siguientes inconvenientes:

1. Se tienen que crear tantas clases como elementos tenga el diagrama de Gantt.
2. La creación de objetos es directa, lo cual se observa al crearlos con la directiva new en main()
3. Hay una clara diferencia entre objetos simples (primitivos) y complejos (compuestos). Es deseable que el cliente trate uniformemente objetos diferentes.
4. No es fácil añadir nuevos componentes. Por cada nuevo componente se tiene(n) que escribir su(s) clase(s) y modificar las clases que dependen de él.

En la parte tres de éste trabajo, se emplea el enfoque de los patrones de diseño para re-estructurar la implementación obtenida y evitar estos inconvenientes.

Capítulo 3

Patrones de diseño implementados

Capítulo 3. Patrones de diseño implementados

Patrones de Estructura

El objetivo de los patrones de estructura es diseñar la arquitectura de un sistema aplicando la composición de clases y objetos. Se subclasifican en dos vertientes:

- **De alcance estático.** Se basan en la herencia para adaptar clases.
- **De alcance dinámico.** Se basan en la composición de objetos para ensamblar componentes.

Los patrones de diseño que corresponden a esta clasificación son los siguientes:

- **De alcance estático:**
  - **Adaptador.** Utiliza la herencia simple para adaptar la interfaz de una clase en otra que el cliente espera.
- **De alcance dinámico:**
  - **Adaptador.** Utiliza la composición de objetos para adaptar la interfaz de una clase en otra que el cliente necesita.
  - **Separador.** Separa la abstracción de la implementación de una clase para que ambas varien independientemente.
  - **Compositor.** Compone objetos en estructuras jerárquicas de árbol para representar relaciones de agregación.
  - **Decorador.** Anexa responsabilidades dinámicas a un objeto.
  - **Fachada.** Proporciona una interfaz común a un conjunto de interfaz en un sistema.
  - **Fragmentador.** Permite definir objetos de diferente granularidad y tamaño.
  - **Substituto.** Proporciona una imagen de un objeto para que el original pueda ser incorporado en demanda.

De los patrones mencionados, **compositor** ensambla objetos para representar jerarquías todo-parte (part-whole), que es precisamente la arquitectura del caso de estudio "el ensamble de un jean", que se desarrolló en la parte dos de éste trabajo.

En esta sección, se describe el patrón compositor siguiendo el formato básico presentado por Gamma et al [*3*], excepto por pequeños detalles, que fueron necesarios para enriquecer la explicación del patrón en la re-estructuración del diseño del caso de estudio.

Compositor

Objetivo. Componer y/o ensamblar objetos en estructura de árbol para representar jerarquías "part-whole". Compositor permite que los clientes traten uniformemente objetos simples y objetos compuestos.

Motivación. La construcción de los componentes de un sistema, por lo general, sigue un proceso jerárquico, en el cual los niveles superiores se forman de niveles inferiores. Sin embargo, la estructura organizativa suele tratar en forma diferente los componentes que son ensambles y los que no lo son, lo

cual ocasiona conflictos en el diseño de sistemas. Es deseable que el desarrollador trate en forma semejante objetos de cualquier tipo.

El patrón compositor describe como usar la composición recursiva para evitar tales ambiguedades.

Analogía con el caso de estudio. La descomposición funcional del jean, figura 2.6, corresponde a una estructura jerárquica, en la cual los componentes (ensambles) de niveles superiores, se integran de niveles inferiores. Cada nivel define componentes que pueden o no ser ensambles. La línea de producción del jean, motiva que las piezas que son ensambles tengan un tratamiento diferente de los que no lo son, lo cual incrementa la complejidad en los procesos de producción.

Estructura. La clave en este patrón es la definición de las clases Componente y Compositor. Componente es una clase abstracta que define atributos y métodos comúnes a todos los elementos en la jerarquía. Compositor agrupa y/o compone los elementos en cada nivel de la estructura jerárquica. La estructura se observa en la figura 3.1.

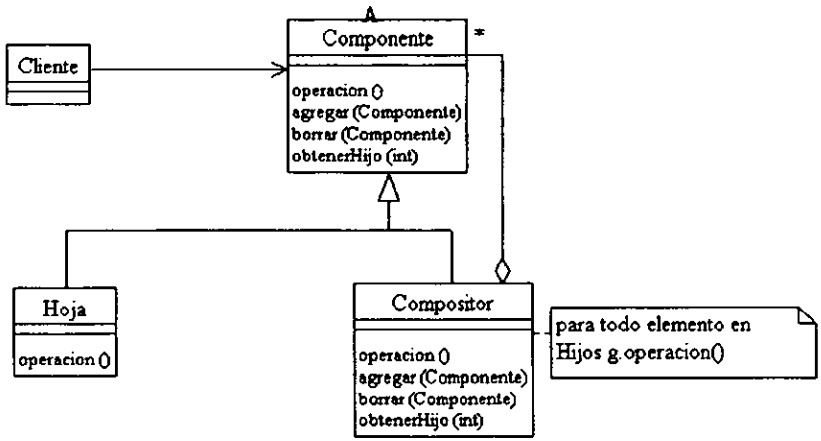

Figura 3.1. Estructura del Patrón Compositor.

Variantes para la implementación. Las más relevantes son las siguientes (las que se implementaron se señalan):

1. **Referencia explícita de los hijos a los padres.** La referencias de los hijos a los padres facilita la navegación en la estrutura de árbol en ambos sentidos, top-down y bottom-up. Sin embargo, la referencia no es fácil de implementar. Lo más común es definirla en la clase Componente. El patrón Cadena de Responsabilidades se basa en éstas referencias.
2. **Compartir componentes.** Cuando se permite la herencia multiple, la propagación de mensajes bottom-up y top-down conduce a ambiguedades. El patrón Fragmentador proporciona una solución permitiendo tener varios padres. En el caso del jean, Broche y B_T tienen varios padres.

sin embargo cada rama de la estructura jerárquica requiere de un objeto del tipo Broche y B_T. Para hacer la implementación del patrón, en el método main(), se crean tantos objetos como sean necesarios de Broche y B_T .

3. **Máximizar la interfaz de Componente.** Compositor y Hoja heredan de Componente, pero Compositor puede estar formado por clases Hoja y Componente. En Componente se definen los métodos comúnes a los hijos Hoja y Compositor, lo que puede conducir a que Hoja o Compositor tengan métodos que no utilizen. Una solución es ver a Hoja como una clase Componente que no tiene hijos, y dejar que Compositor redefina los métodos en conflicto. En el caso del jean, aunque se definieron pocos métodos, Compositor redefine los métodos en conflicto, por ejemplo: coser().
4. **Donde declarar los métodos principales.** Existen dos enfoques: el primero llamado de transparencia ocurre cuando una subclase hereda los métodos de su clase padre, aunque en la clase hija no tengan sentido o no se usen, por ejemplo, si se declaran agregar() y borrar() en Componente, en Hoja no se utilizan, ya que por definición una hoja no tiene descendencia. El segundo se llama de seguridad y se da cuando sólo se declaran los métodos en la subclase que los utiliza, por ejemplo, si se definen agregar() y borrar() en Compositor, Hoja y Compositor no tendran la misma interfaz. En el caso del jean, se prefirio la opción de seguridad.
5. **Donde definir los componentes.** La forma más usual es definir un arreglo en Compositor que contenga los componentes; para el caso del jean parte[] es dicho arreglo.
6. **Ordenar los hijos.** Dependiendo de la aplicación, algunas veces es deseable tener ordenados los hijos en la estructura de árbol. La navegación en la estructura se suele hacer con el patrón Iterador. En el caso del jean, Iterador se implementa por la clase Costurera.
7. **Ocultar para un mejor desempeño.** Cuando constantemente se está navegando en la estructura es conveniente ocultar la implementación de los métodos que no se utilizan. La interacción únicamente se da con los métodos que proporcionan la funcionaliad requerida. Este tipo de interfaz son dificiles de diseñar, ya que se tiene que invalidar la implementación de los métodos no deseados en la navegación.
8. **Quien debe borrar los componentes.** En lenguajes sin recolector de basura, Compositor debe ser el responsable. Una excepción es cuando Hoja es inmutable. Java tiene implícito el recolector de basura, por lo tanto, en el caso de estudio no se borra ningun elemento de la jerarquía.
9. **Cuál es la mejor estructura para almacenar los componentes.** La estructura elegida depende de la eficiencia requerida. En el caso del jean se usó un arreglo.

Patrones relacionados. Los patrones que se relacionan son: Cadena de Responsabilidades, Decorador, Fragmentador, Iterador y Visitador. Los dos últimos, se estudian en la sección de patrones de desempeño.

Implementación del patrón Compositor. En el diagrama de clases del *caso de estudio* "el ensamble de un jean", se observa que se han definido tantas clases como componentes en la jerarquía. Los ensambles en cada nivel jerárquico indican quienes son los candidatos a objetos compuestos. Los componentes que no tienen descendencia son los objetos simples o primitivos.

Haciendo una analogía de la estructura del patrón de la figura 3.1, se obtiene el diagrama de la figura 3.2. En él se observa que los objetos primitivos son P_D_T, Ex_D_T,....y P_Cierre. El único objeto compuesto es Ensamble. Ensamble representa los componentes: Jean01, P_I_D_S, P_D_S, P_D_S_T,.., y P_I_S_D.

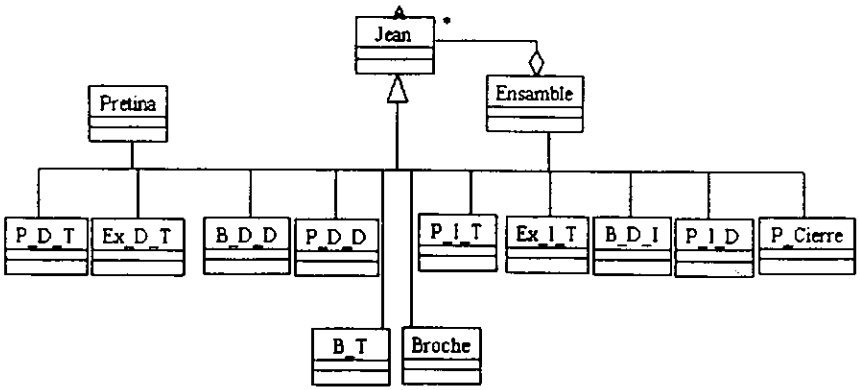

Figura 3.2. Implementación del Patrón Compositor.

El código del diseño de la figura 3.2 es el siguiente. La clase Jean es abstracta y define, entre otros, el método coser() que todas las subclases heredan.

Jean

```
abstract class Jean
{
protected boolean cosido = false;
protected String nombre;
...
public void coser()
{
cosido = true;
}
...
}
```

La clase Ensamble hereda de Jean y define como atributos un arreglo dinámico, cuyo tamaño es el número de elementos (piezas) que integran cada ensamble. El método coser() se redefine.

Ensamble

```
public class Ensamble extends Jean
{
private int max; // maximo tamaño del arreglo
private int numPiezas; // piezas del ensamble
private Jean [] parte; // arreglo de piezas de cada ensamble
...
```

```
public void ensamblar(Jean j)
{
...
parte[numPiezas++] = j;
...
}
...
public void coser()
{
for(...)
{
parte[i].coser();
}
...
}
}
```

El método ensamblar compone objetos, es decir, los objetos que integran cada ensamble son almacenados en parte[]. Cada objeto compuesto define su propio arreglo. La invocación del método coser() en cada objeto compuesto, provoca que cada integrante invoque a su vez su propio método coser(), la reacción en cadena es un ejemplo de polimorfismo por inclusión, ver *glosario*. La comunicación entre los objetos se observa en la figura 3.3.

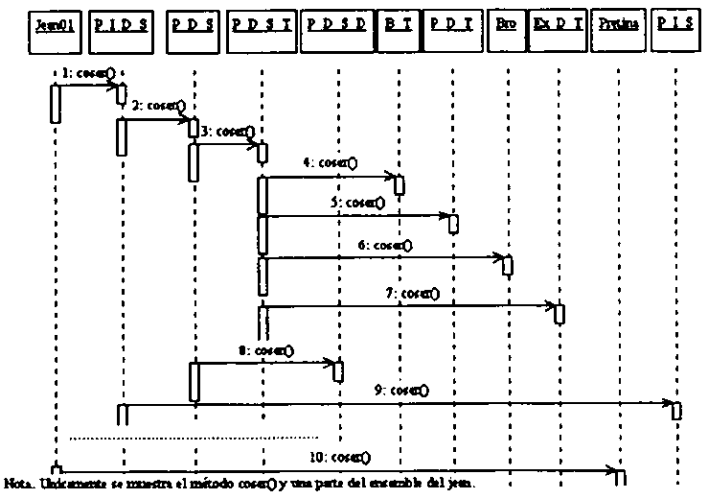

Figura 3.3 . Diagrama de Secuencias de la implementación del patrón Compositor.

Mientras que en la implementación obtenida en la parte 2 de este trabajo cada ensamble representaba una clase, en la implementación aplicando el patrón compositor los ensambles son representados por una sola clase Ensamble. El cliente trata en forma semejante los objetos compuestos y los simples.

Por último, en el método main() se crean los objetos simples y compuestos. Se crean tantos objetos

compuestos como ensambles se tengan definidos en la jerarquía de árbol. Cada ensamble reserva el espacio del tamaño de sus integrantes, y después uno a uno los almacena.

El código del ensamble del jean se encuentra disponible en línea: *PatronComposite.java.*

Ventajas:

1. El cliente trata en forma semejante objetos simples y compuestos.
2. Los objetos compuestos quedan representados por una sola clase Ensamble.
3. Se fomenta la generación de mensajes en cadena, empleando el polimorfismo.
4. Se facilita la agregación de componentes.
5. Es aplicable a todo tipo de objetos, ya que, en la vida real los objetos son complejos.

Desventajas:

1. No es fácil hacer la analogía de la estructura de Compositor a un dominio específico.
2. No es fácil diseñar la interfaz de la clase abstracta, ya que se tienen que identificar los métodos comunes de los hijos. La abstracción es uno de los problemas más serios en la programación orientada a objetos. Una buena abstracción conduce a la reutilización de componentes
3. Compositor no distingue entre Clasificación y Generalización (herencia), ni Agregación y Composición [*4*].

Más adelante, estudiaremos como mejorar la estructura del caso de estudio con la implementación de los siguientes patrones:

- **Creación de objetos:** Método de Fabricación, Fábrica Abstracta y Constructor.
- **Desempeño de objetos:** Método de Plantilla, Iterador y Visitador.

Patrones de Creación

El objetivo de los patrones de creación es el diseño de interfaces para tener mayor flexibilidad en la forma de instanciar objetos. Se subclasifican en dos ramas:

- **De alcance estático.** Se basan en la herencia para diferir la creación de objetos a las subclases.
- **De alcance dinámico.** Se basan en la herencia para transferir la creación de objetos a otras clases.

Los patrones de diseño que corresponden a esta clasificación son los siguientes:

- **De alcance estático:**
  - **Método de Fabricación.** Define una interfase que contiene un método abstracto que las subclases implementan para instanciar objetos. Las subclases deciden el tipo de los objetos.
- **De alcance dinámico:**
  - **Fábrica Abstracta.** Define una interfase para crear familias de objetos sin especificar sus clases concretas.
  - **Constructor.** Define una interfase para crear familias de objetos complejos y/o compuestos que no depende de su estructura interna, para que el mismo proceso de construcción pueda crear varias representaciones.
  - **Prototipo.** Define una interfase que es un prototipo para crear objetos.
  - **Individual.** Define una forma para que una clase solo tenga una instancia.

La creación de objetos del caso de estudio, el ensamble de un jean, se re-estructuró con la implementación de los siguientes patrones: Método de Fabricación, Fábrica Abstracta y Constructor.

Método de Fabricación

Objetivo. Definir una interfase para crear un objeto y permitir que las subclases decidan que clase instanciar.

Motivación. Consideremos un sistema cuya tarea principal es mostrar los archivos que las aplicaciones manipulan. Las aplicaciones y los archivos son clases abstractas, y una aplicación instancia un número determinado de archivos. Una aplicación tiene como responsabilidad abrir archivos pero no sabe que tipo de archivos, lo cual crea un dilema.

El patrón Método de Fabricación ofrece una solución al implementar un método en cuyo cuerpo se invoca una operación abstracta que las subclases implementan. Una vez que se crea una aplicación, cada una de ellas instancia su propia versión.

Analogía con el caso de estudio. En la línea de producción del jean, el elemento esencial son las costureras. La complejidad de la unión de las piezas que componen cada ensamble, motiva que las tareas se especializen. Así, en nuestro caso, se determina que debe de haber una costurera por cada ensamble. Sin embargo, el nombre de la costurera es desconocido hasta el preciso momento es que son necesarios sus servicios. Los ensambles se deben de coser, sin embargo, no se sabe quien los va a coser.

Estructura. La estructura del patrón consiste de dos clases abstractas Producto y Productor. La primera define una familia de productos, mientras que la segunda crea los mismos. Clases concretas

implementan los métodos abstractos, la estructura se observa en la figura 3.4.

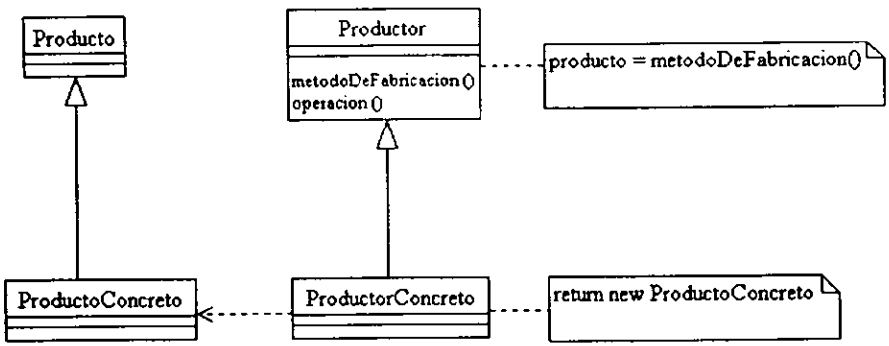

Figura 3.4. Estructura del Patrón Método de Fabricación.

Variantes para la implementación. Las más relevantes son las siguientes (las que se implementaron se señalan):

1. **Como definir la clase.** La clase donde se define el metodoDeFabricacion(), que es el que da el nombre al patrón, puede ser abstracta y/o concreta. Si es concreta, el proceso de creación de objetos es aislado, lo que permite que las subclases redefinan la forma de crearlos. Si es abstracta, las subclases implementan metodoDeFabricacion(), el cual tiene que ser abstracto.
   En el caso de estudio se implementaron las dos versiones, el primer caso es el método asignarCosturera() de la clase concreta Ensamble, en la implementación del patrón compositor; en el segundo caso se declara el método abstracto crearIterador(), en la implementación del patrón iterador. En ambos casos, los métodos señalados iteran sobre una estructura específica.
2. **Parametrizar metodoDeFabricacion().** La creación de objetos se puede discrecionar obedeciendo a condiciones dependientes de la funcionalidad deseada. El framework Unidraw [*30*] utiliza este enfoque.
3. **Dependencias de los lenguajes.** Smalltalk utiliza metodoDeFabricacion() para regresar la clase del objeto que se va a instanciar. En C++ metodoDeFabricacion() es siempre una función virtual.
4. **Plantillas.** metodoDeFabricacion() puede ser implementado como una plantilla, lo cual evita que las subclases decidan que clase instanciar.

Patrones relacionados. Fábrica Abstracta, Método de Plantilla y Prototipo.

Implementación del patrón Método de Fabricación. Se implementaron dos versiones. El primero corresponde al caso de estudio "el ensamble de un jean", con la implementación del patrón compositor; el segundo es la implementación del patrón iterador con las estructuras abstractas: árbol binario, pila y arreglo. El diagrama de clases de la implementación se observa en la figura 3.5.

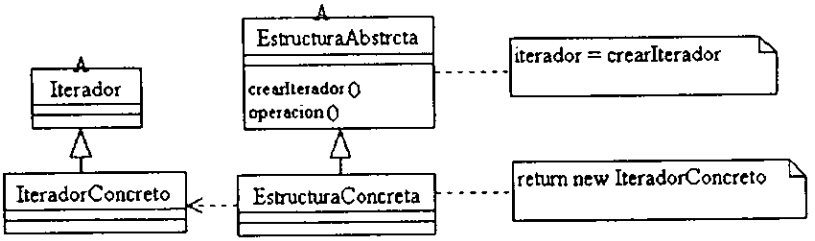

Figura 3.5a. Estructura del Patrón Método de Fabricación, implementado con una estructura abstracta y su iterador.

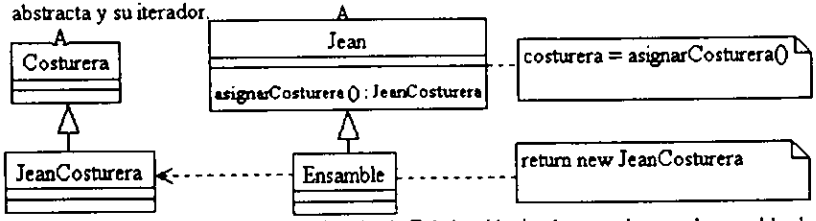

Figura 3.5b. Estructura del Patrón Método de Fabricación, implementado con el ensamble de un jean.

En el ensamble del jean, los métodos coser() y revisar() invocan asignarCosturera() en la clase concreta Ensamble. Ambos métodos instancian la clase Costurera.

Ensamble

```
public class Ensamble extends Jean
{
.....
public void coser()
{
Costurera K = asignarCosturera(); // metodoDeFabricacion();
for(k.primero(); !k.heTerminado(); k.siguiente())
{
((Jean)k.parteaCoser()).coser();
}
}
...

public Costurera asignarCosturera()
{
return new JeanCosturera(this);
}
....

public boolean revisar(Object elemento)
{
Costurera K = asignarCosturera();
```

```
for(k.primero(); !k.heTerminado(); k.siguiente())
{
if( ((Jean)K.parteaCoser()).revisar(elemento) )
return true;
}
...
}
```

En el método coser(), se instancia la clase JeanCosturera para crear una costurera especializada en la costura de jeans. JeanCosturera es una subclase de Costurera.

En el método revisar(), se instancia la clase JeanCosturera para crear una costurera que se especializa en el control de calidad.

El método asignarCosturera puede elegir de un conjunto de costureras, lo que ilustra claramente que el ensamble denotado por "this" es cosido por una costurera especifica

En este ejemplo, el tipo de parámetro de revisar(..) es declarado como Object para designar un dato genérico. Java define dos tipos de datos, primitivo y de referencia. Los primitivos corresponden a int, double, float, etc., los de referencia a los arreglos y cadenas [*24*].

En el desarrollo de las estructuras abstractas, se declara una clase abstracta EstructuraAbstracta, la cual implementa el método buscar().

EstructuraAbstracta

```
public abstract class EstructuraAbstracta
{
...
protected boolean buscar(Object elemento)
{
Iterador j = crearIterador();
for(j.primero(); !j.heTerminado(); j.siguiente())
{
if( (j.elementoActual() != null) &&
(j.elementoActual().equals(elemento.toString()) ))
return true;
}
return false;
}

protected abstract Iterador crearIterador();
}
```

Pila, Arreglo y Arbol Binario son estructuras concretas, subclases de EstructuraAbstracta, cada una crea su propio iterador al invocar crearIterador().

En las dos implementaciones mostradas, se están usando otros patrones: compositor, método de plantill

e iterador, lo cual ilustra el echo de que un patrón raras veces existe aislado.

El código de ambas implementaciones se encuentra disponible en línea: *ensamble del jean*, y *estructuras abstractas*.

En la implementación de los patrónes Fábrica Abstracta y Constructor, que más adelante se estudian, también se empléa Método de Fabricación.

Ventajas:

1. Aporta una mayor flexibilidad para la creación de objetos.
2. Ligar jerarquías diferentes y en paralelo. En la implementación de las estructuras abstractas, la jerarquía de las estructuras es usada junto con su iterador.

Desventajas:

1. El método metodoDeFabricacion() se clasifica como de alcance estático, lo que significa que su implementación depende de la herencia de clases. En una estructura jerárquica como la del ensamble del jean, los objetos primitivos no utilizan metodoDeFabricacion(), lo cual favorece la seguridad pero no la transparencia. En el caso de estudio se favoreció la seguridad sobre la transparencia.

Patrón Fábrica Abstracta

Objetivo. Definir una interfase para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

Fábrica abstracta es una generalización del patrón método de fabricación para crear familias de objetos.

Motivación. Consideremos un sistema gráfico que manipula elementos como botones, diálogos, messageBox, radioButtons, etc. Para garantizar la portabilidad del sistema en varias arquitecturas, el diseño debe ser lo más amplio y flexible posible.

El patrón Fábrica Abstracta ofrece una solución al diseñar una interfase que consiste de operaciones abstractas que las subclases implementan. Cada subclase corresponde a un arquitectura diferente. Los elementos de todas las familias, se organizan también en clases abstractas, una por cada tipo de elementos que tengan en común.

Analogía con el caso de estudio. Consideremos tres familias de jeans: clásico, acampanado y estretch. El proceso de producción es semejante en los tres casos, no así su arquitectura. Los elementos que no constituyen ensambles son comúnes y por tanto, suceptibles de clasificarse por tipo de elementos en clases abstractas, sin embargo, los ensambles si pueden variar. Cada familia es responsable de la creación de sus elementos y de la composición de sus ensambles.

Estructura: La estructura del patrón define dos tipos de clases principales: la clase abstracta FabricaAbstracta que tiene métodos abstractos para crear productos y las clases abstractas que

representa los productos. Las fábricas concretas implementan sus propios métodos de fabricación de productos. Cada fábrica concreta corresponde a una familia de productos. La estructura del patrón se observa en la figura 3.6.

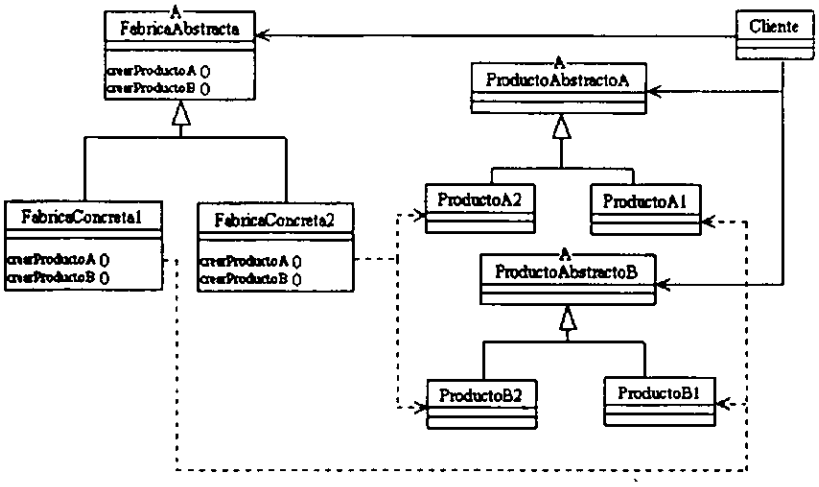

Figura 3.6 . Estructura del patrón Fábrica Abstracta.

Variantes para la implementación. Las más relevantes son las siguientes (las que se implementaron se señalan):

1. **Una instancia por cada fábrica.** Si únicamente se necesita una instancia de una fábrica concreta, la implementación se realiza con el patrón individual.
2. **Creación de los productos.** Por lo general cada método abstracto de FabricaAbstracta es un ejemplo del patrón método de fabricación. En el caso de estudio, cada método que instancia un elemento y/o pieza del jean es un método de fabricación. Si se tienen muchas familias de productos, conviene clasificarlos utilizando el patrón Prototipo.
3. **Definiendo más fábricas.** Cuando se desean agregar nuevos productos, la interfase de FabricaAbstracta se tiene que modificar; una posible solución es identificar con un parámetro el objeto que se va a crear. Este es el enfoque en que se basa el patrón prototipo. En el caso de estudio, los productos son limitados, sin embargo, agregar un nuevo producto significaría cambiar las interfaces de FabricaAbstracta y de todas las clases dependientes. Otro enfoque sería el de agregar responsabilidades con el patron Visitador. El patrón Visitador se estudia más adelante.

Patrones relacionados. Método de Fabricación, Prototipo, Individual y Separador.

Implementación del patrón Fábrica Abstracta. Hipotéticamente se consideran tres familias de jeans: Jean Clasico, Jean Acampanado, y Jean Stretch. La clase abstracta FabricaAbstracta define métodos abstractos por cada elemento del jean. Las tres familias comparten las mismas piezas, únicamente se distinguen en características que pueden ser modeladas con los atributos. Se tienen tres fábricas concretas, una para cada familia. Los productos, son las piezas del jean, las cuales se han agrupado en

clases similares. Los métodos comúnes de éstas clases se han subido de nivel en la clase abstracta Jean. El diseño de la implementación se observa en la figura 3.7.

La interfase de la clase FabricaAbstracta que se observa en la figura 3.7, define únicamente la creación de objetos simples; el tratamiento de los objetos complejos y/o ensambles se han postergado hasta la siguiente sección cuando se estudie el patrón Constructor.

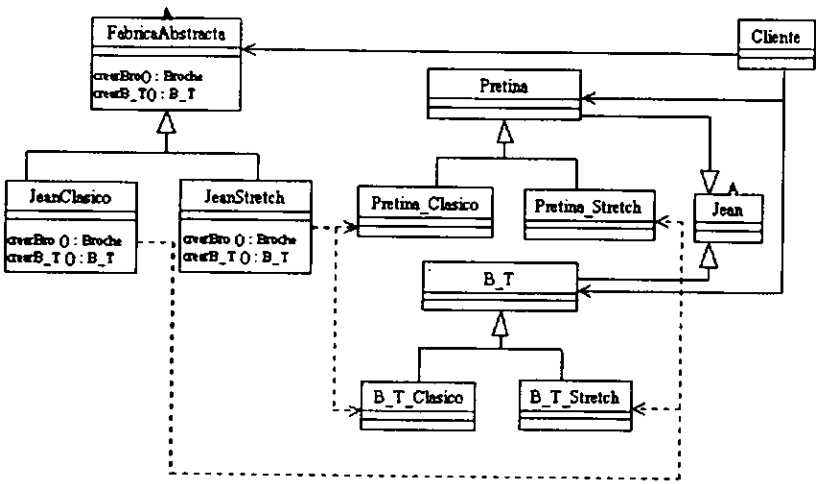

Figura 3.7. Estructura del Patrón Fábrica Abstracta implementado con familias de jeans.

La dinámica de la implementación de la pieza Pretina se describe enseguida; el código para el resto de los elementos es similar.

Piezas del jean

```
abstract class Jean
{
....
public void coser()
{
cosido = true;
}
...
}
```

// Familia de clases de Pretina

```
class Pretina extends Jean
{
...
public void coser()
```

```
{
super.coser();
System.out.println(super.miNombreEs() + "---->cosido");
}
...
}
```

// Pretina del jean clásico

```
class Pretina_Clasico extends Pretina
{
...
public void coser()
{
super.coser();
}
...
public metodoxJean_Clasico(..)
{
...
}
}
```

// Pretina del jean stretch

```
class Pretina_Stretch extends Pretina
{
..
public void coser()
{
super.coser();
}
...
public metodoxJean_Stretch(..)
{
...
}
}
```

// Pretina del jean acampanado

```
class Pretina_Acampanado extends Pretina
{
...
public void coser()
{
super.coser();
}
```

```
...
public metodoxJean_Acampanado(..)
{
...
}
}
```

Fábricas

```
public abstract class FabricaAbstracta
{
...
public abstract Pretina crearPretina();
....
}
```

Fábrica de jean clásico

```
class JeanClasico extends FabricaAbstracta
{
...
public Pretina crearPretina();
{
return new Pretina_Clasico(...);
}
...
}
```

Fábrica de jean stretch

```
public class JeanStretch extends FabricaAbstracta
{
...
public Pretina crearPretina();
{
return new Pretina_Stretch(...);
}
...
}
```

Fábrica de jean acampanado

```
public class JeanAcampanado extends FabricaAbstracta
{
..
public Pretina crearPretina()
{
```

```
return new Pretina_Acampanado(...);
}
...
}
```

La creación de los objetos primitivos es responsabilidad de cada familia, mientras que los compuestos se ensamblan una vez creados los primeros.

FabricaAbstractaCompositor

```
public class FabricaAbstractaComponedor
{
....
public static void main(String args[])
throws java.io.IOException
...
FabricaAbstracta fabricaJeanClasico = new JeanClasico();
Pretina pretina = fabricaJeanClasico.crearPretina();
....
Ensamble p_d_s_t = new Ensamble(4,"P_D_S_T_Clasico");

p_d_s_t.ensamblar(b_t);
p_d_s_t.ensamblar(p_d_t);
p_d_s_t.ensamblar(bro);
p_d_s_t.ensamblar(ex_d_t);

}
```

La diferencia de éste patrón Fábrica Abstracta con el patrón anterior Método de Fabricación, es que el segundo instancia objetos individuales, mientras que el primero objetos de una familia. Ambos regresan el control "inmediatamente" despues de instanciar el objeto, lo cual se observa con la creación de los objetos primitivos. El inconveniente con los objetos compuestos o ensambles es que apriori se deben de crear sus elementos primitivos integrantes, es decir, existe un proceso intermedio antes de constituir un ensamble. El siguiente patrón "constructor" se aboca a la creación de objetos que involucran un proceso

El código de la implementación se encuentra disponible en línea: *fábrica abstracta*

El diagrama de secuencias de la implementación del patrón se observa en la figura 3.8.

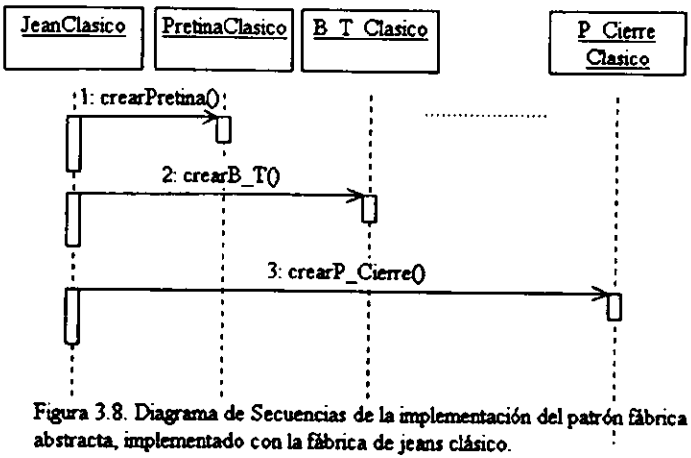

Figura 3.8. Diagrama de Secuencias de la implementación del patrón fábrica abstracta, implementado con la fábrica de jeans clásico.

Ventajas:

1. Aisla clases concretas. El proceso de instanciar objetos es encapsulado en una sóla clase "Fábrica". El cliente manipula instancias únicamente a traves de interfaces abstractas. Los nombres de los productos concretos no aparecen en el código del cliente.
2. Es relativamente fácil cambiar de fábrica, lo cual se aprecia en main(). Es factible la configuración de diferentes productos.
3. Se promueve la consistencia y homogeneidad entre productos

Desventajas:

1. El diseño de nuevos productos es difícil; se tendría que modificar la interfase de FabricaAbstracta. Prototipo ofrece una solución al identificar con un parámetro los objetos creados. Visitador aporta otra solución al agregar responsabilidades en tiempo de ejecución.

Patrón Constructor

Objetivo. Separar la construcción de un objeto complejo y/o compuesto de su representación, para que el mismo proceso de construcción pueda crear diferentes representaciones.

Motivación. Consideremos un sistema cuya tarea principal es convertir texto en diferentes formatos. Dado que el número de formatos es ilimitado, el diseño del sistema debe ser tal que su funcionalidad se mantenga independiente de los formatos existentes.

El patrón Constructor ofrece una solución al separar la funcionalidad del objeto, de su estructura interna. La funcionalidad es un invariante de la arquitectura.

Analogía con el caso de estudio. La línea de producción del jean, específicamente las plantas de ensamble, constantemente están cambiando. Los planes de producción, por otra parte, siguen políticas más estables. La incorporación en la línea de producción de un nuevo ensamble o un estilo diferente a los existentes, puede ocasionar cambios en las políticas de producción. La dependencia en ambos procesos puede ocasionar problemas en el desenvolvimiento del sistema textil.

Mientras que el patrón Método Abstracto es un caso particular de Fábrica Abstracta, Constructor se puede considerar como la generalización para crear objetos complejos de los dos primeros. De echo, la implementación lograda hace énfasis en la reutilización de ambas soluciones.

Estructura. Dos clases concretas caracterizan el patrón constructor: Director y Constructor. Director dirige el proceso de producción. Constructor determina la forma en que se construyen los objetos. Ambas clases dirigen y coordinan la producción de objetos. La estructura se observa en la figura 3.9.

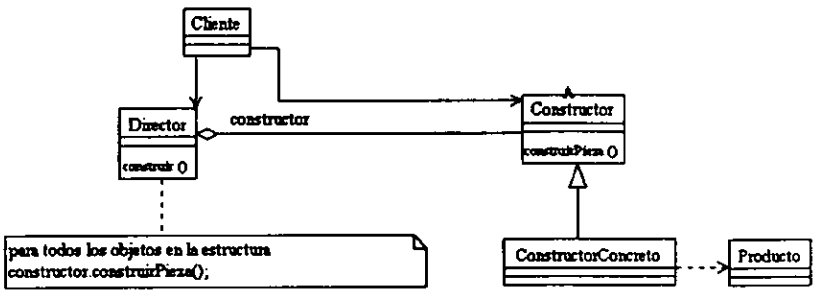

Figura 3.9. Estructura del patrón Constructor.

Variantes para la implementación. Las más relevantes son las siguientes (las que se implementaron se señalan):

1. **Interfase de la clase Constructor:** Constructor crea objetos compuestos paso a paso; su interfase se debe definir lo más general posible para que pueda construir casi cualquier tipo de objeto. En el caso de estudio, el patrón constructor crea objetos complejos y se apoya en Fábrica Abstracta y método de fabricación para construir objetos simples.
2. **Diseño de la clase de los productos:** Comunmente el cliente configura al director con un constructor adecuado, así mismo, conoce que subclase de constructor esta instanciando productos. En el caso de estudio, el cliente configura al constructor con la fábrica de jeans adecuada. El director únicamente dirige la producción.

Patrones relacionados: Fábrica Abstracta, Componedor y Método de Fabricación.

Implementación del patrón Constructor: La estructura de la implementación se observa en la figura 3.10. Notese que no hay una fuerte cohesión entre las clases Director y Constructor. Director utiliza la

relación usando [9], para expresar que esta utilizando los servicios de Constructor. La clase Constructor, por otra parte, tiene como atributo un objeto del tipo Fábrica Abstracta, el cual define una interfase para crear familias de objetos. A su vez, los métodos que integran dicha interfase son ejemplos del patrón Método de Fabricación. El diagrama de la implementación se aprecia en la figura 3.10. La comunicación de los objetos se observa en el diagrama de secuencias de la figura 3.11.

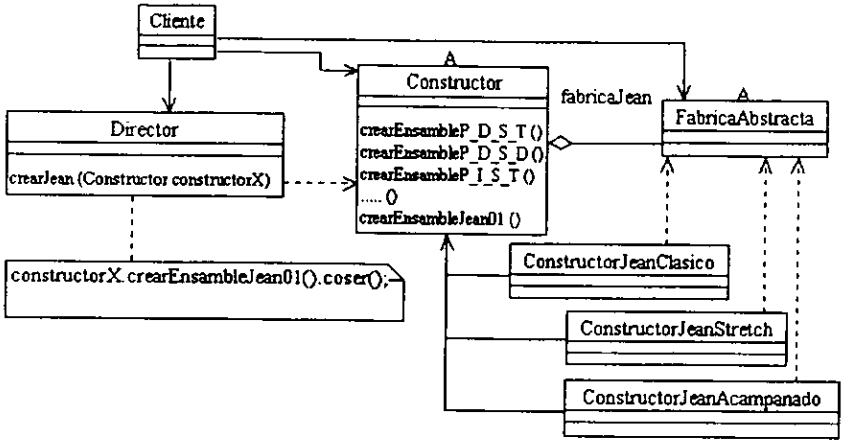

Figura 3.10 Estructura del Patrón Constructor implementado con el ensamble de una familia de jeans..

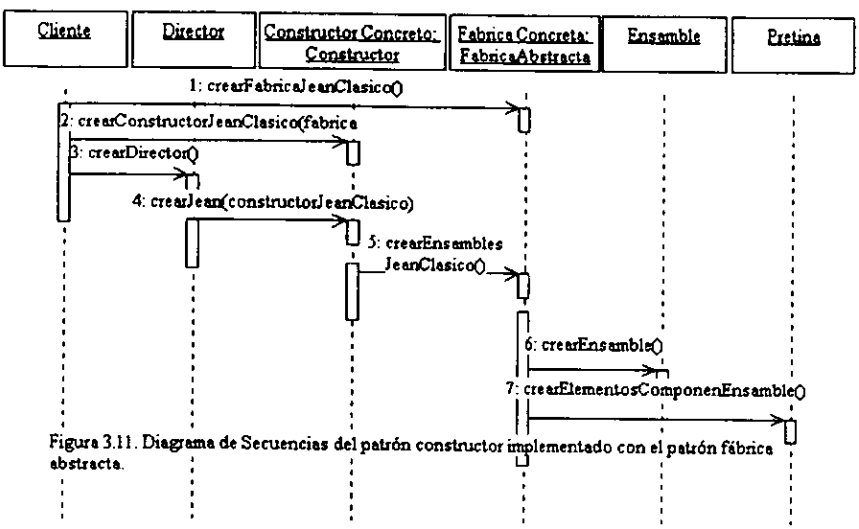

Figura 3.11. Diagrama de Secuencias del patrón constructor implementado con el patrón fábrica abstracta.

La interfase de constructor especifica métodos generales para crear ensambles. Cada constructor concreto personaliza la forma de crearlos, para lo cual instancia la fábrica correspondiente. fabricaJean

es una implementación del patrón Fábrica Abstracta.

Constructor

```
class Constructor
{
protected FabricaAbstracta fabricaJean;
public Constructor(FabricaAbstracta fabricaJeanX)
{
fabricaJean = fabricaJeanX;
}

public abstract Ensamble crearEnsambleP_D_S_T();
public abstract Ensamble crearEnsambleP_D_S_D();
....
}
```

ConstructorJeanClasico

```
public class ConstructorJeanClasico extends Constructor
{
....
public Ensamble crearEnsambleP_D_S_T()
{
B_T b_t = fabricaJean.crearB_T();
P_D_T p_d_T = fabricaJean.crearP_D_T();
Broche bro = fabricaJean.crearBro();
Ex_D_T ex_d_t = fabricaJean.crearEx_D_T();

Ensamble p_d_S_t = fabricaJean.crearEnsambleP_D_S_T();

p_d_S_t.ensamblar(b_t);
p_d_s_t.ensamblar(p_d_t);
p_d_s_T.ensamblar(bro);
p_d_s_t.ensamblar(ex_d_T);
return p_d_s_t;
}
....
}
```

La construcción de los ensambles se aprecia en crearEnsambleP_D_S_T(); mientras que Constructor define un método para cada ensamble, el constructor concreto lo implementa. Las partes que componen cada ensamble se crean con una fábrica especifica. El cliente elige la fábrica de jeans y configura el constructor.

La interfase de Director es bastante sencilla, sólo tiene un método que invoca la creación de ensambles

en el constructor.

Director

```
public class Director
{
public Director(){ }
public void crearJean(Constructor constructorX)
{
constructorX.crearEnsambleJean01().coser();
}
}
```

En el método main(), el cliente determina el tipo de jean que desea, la creación de los objetos ha quedado oculto.

ConstructorFabricaAbstractaComponedor

```
public class ConstructorFabricaAbstractaComponedor
{
public static void main(String args[])
throws java.io.IOException
{
FabricaAbstracta fabricaJeanClasico = new JeanClasico();
Constructor constructor = new Constructor(fabricaJeanClasico);
Director director = new Director();

director.crearJean(constructor);
}
}
```

El código de la implementación se encuentra disponible en línea: *constructor.*

Ventajas:

1. Aisla los productos. Sean estos complejos o simples, el cliente no tiene acceso a ellos. Constructor se puede configurar con cualquier fábrica.
2. Promueve la separación de la construcción y la representación del objeto. Si constructor se diseña como una clase abstracta, cada subclase es una forma diferente de construir objetos. En este sentido, el cliente no tiene acceso a la estructura interna de los objetos. En el caso de estudio, las familias de jeans consideradas tienen el mismo proceso de construcción, por lo que, ésta ventaja no se aprecia.
3. Control del proceso de producción. Mientras que el director dirige el proceso de construcción, el constructor lo coordina.
4. Se promueve la consistencia y homegeneidad entre productos.

Desventajas:

1. La interfase de Constructor define los productos que se van a fabricar, si los productos deseados no corresponden a ésta interfase, es necesario modificarla. Prototipo ofrece una solución al identificar con un parámetro los objetos creados. Visitador aporta otra solución al agregar responsabilidades en tiempo de ejecución.

Patrones de Desempeño

El objetivo de los patrones de desempeño es el diseño de interfaces para promover la interacción de clases y objetos, y la distribución de responsabilidades. Se subclasifican en dos ramas:

- **De alcance estático.** Se basan en la herencia para definir un comportamiento que las subclases pueden redefinir.
- **De alcance dinámico.** Describen diversas formas para facilitar la comunicación entre objetos.

Los patrones de desempeño que corresponden a esta clasificación son los siguientes:

- **De alcance estático:**
  - **Interpretador.** Utiliza la herencia para definir un interprete de la gramática de un lenguaje, para que las subclases definan su propia representación.
  - **Método de Plantilla.** Define un algoritmo en términos de métodos abstractos que las subclases implementan.
- **De alcance dinámico:**
  - **Cadena de Responsabilidades.** Define una interfase común entre una cadena de objetos, para que compartan responsabilidades.
  - **Comando.** Encapsula la responsabilidad de un objeto para parametrizar la solicitud de algun servicio.
  - **Iterador.** Define un objeto para navegar en los elementos de una estructura sin exponer su representación interna.
  - **Mediador.** Define un objeto que encapsula la interacción entre varios objetos. Evita que los objetos se comuniquen directamente, promoviendo la independencia de los mismos.
  - **Temporal.** Sin violar la encapsulación, registra el estado interno de un objeto para que pueda regresar a su estado anterior.
  - **Observador.** Define una relación uno a muchos entre objetos, para que cuando uno cambie, los demás se actualizen automáticamente.
  - **Estado.** Permite que un objeto altere su comportamiento cuando su estado interno cambia.
  - **Estrategia.** Define una familia de algoritmos que se pueden intercambiar dependiendo de las necesidades de los clientes.
  - **Visitador.** Define una funcionalidad adicional representada por una operación que puede ser incorporada en la interfase de otros objetos.

La comunicación entre los objetos del caso de estudio, el ensamble de un jean, se re-estructuró con la implementación de los siguientes patrones: Método de Plantilla, Iterador y Visitador.

Patrón Método de Plantilla

Objetivo. Define el esqueleto de un algoritmo en términos de operaciones abstractas que las subclases implementan.

Motivación: Consideremos un sistema con dos componentes, aplicaciones y archivos. Una aplicación es responsable de ejecutar operaciones en los archivos como abrir y cerrar. Las aplicaciones y los archivos son diseñados como clases abstractas y por tanto no se pueden instanciar. Además, una aplicación

desconoce la clase de archivo en el cual debe operar.

El patrón Método de Plantilla ofrece una solución al implementar el algoritmo de operaciones en términos de clases abstractas, las cuales las subclases implementan para especializar las tareas.

Analogía con el caso de estudio. El control de calidad en las líneas de producción textil es dependiente del proceso de confección de cada estilo. Revisar la confección de los ensambles es un proceso común a todos los estilos de jeans, excepto por pequeños detalles afines de cada cada uno de ellos. El patrón Método de Plantilla ayuda en la definición de un método común revisar() que cada uno de los estilos de jeans modifica para adaptarlo a su propio proceso de manufactura.

Estructura. El patrón se caracteriza por la implementación de un algoritmo en una clase abstracta que las subclases redefinen, figura 3.12.

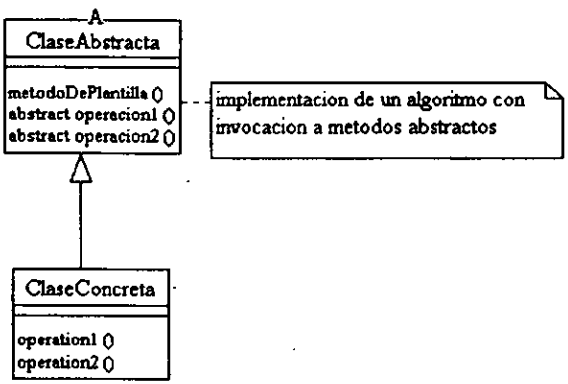

Figura 3.12 Estructura del Patrón Método de Plantilla.

Variantes para la implementación. Las más relevantes son las siguientes (las que se implementaron se señalan):

1. **Visibilidad.** Existen dos enfoques en su implementación. Es de trasparencia si se define el método de plantilla en una clase abstracta y las subclases, si lo utilizan, lo implementan. Es de seguridad si se define en la subclase que únicamente lo utiliza. En las dos implementaciones logradas de éste patrón se aprecian los dos enfoques: de trasparencia con el método buscar() de las estructuras abstractas y, de seguridad con el método revisar() del ensamble del jean.
2. **Invocaciones.** En el cuerpo del método de plantilla se pueden definir las llamadas a variables de otras clases, métodos abstractos y métodos concretos. En el caso de estudio del jean, se invoca el método abstracto revisar(). En el caso de las estructuras abstractas se invoca el método abstracto crearIterador().

Patrones relacionados. Método de Fabricación y Estrategia.

Implementación del patrón Método de Plantilla. En esta sección se describen dos versiones de la implementación del patrón. En la figura 3.13a se observa el diagrama de clases de las estructuras abstractas; el método de plantilla corresponde a buscar(). Define un algoritmo para permitir la navegación en las estructuras arreglo, pila y árbol binario. En la figura 3.13b se observa el método de plantilla revisar() en la clase concreta Ensamble. revisar() simula el proceso que realiza un cliente cuando acepta de conformidad el lote de jeans que solicitó.

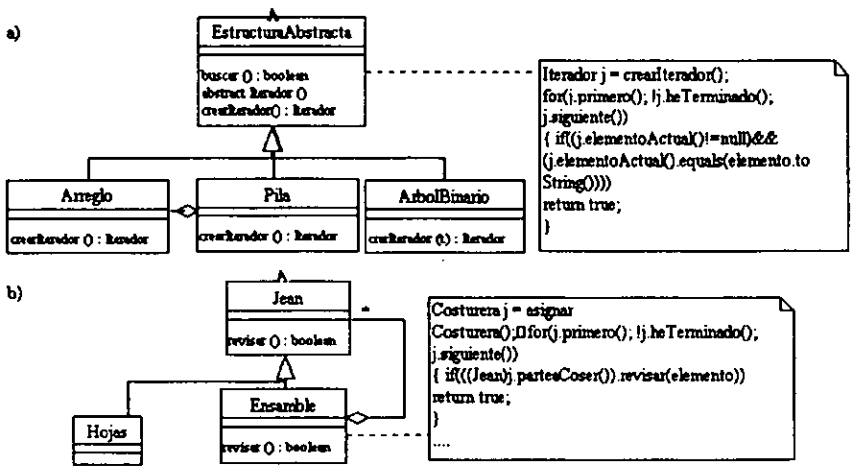

Figura 3.13 . Implementación del patrón Método de Plantilla. revisar() y buscar() con ejemplos de Método de Plantilla.

Ensamble del jean

Ensamble

```
public class Ensamble extends Jean
{
.....
public boolean revisar(Object pieza)
{
Costurera k = asignarCosturera();

for(k.primero(); !k.heTerminado(); k.siguiente())
{
if( ((Jean)k.parteaCoser()).revisar(elemento) )
return true;
}

if( nombre.toString().equals(elemento.toString()) )
```

```
{
System.out.println("El elemento: "+elemento.toString()+" es un .....ensamble");
return true;
}
return false;
}

}
```

revisar() invoca el método asignarCosturera(), que es un ejemplo del patrón método de fabricación. Se destina una costurera por ensamble y se revisan todos los ensambles y/o piezas de que se compone. La estructura en la cual opera método de plantilla corresponde a la del patrón compositor. La llamada a revisar() es recursiva siguiendo una estructura de árbol por ensamble.

Estructuras Abstractas

EstructuraAbstracta

```
public abstract class EstructuraAbstracta
{
.....
protected boolean buscar(Object elemento)
{
Iterador j = crearIterador();
for(j.primero(); !j.heTerminado(); j.siguiente())
{
if( (j.elementoActual() != null) &&
(j.elementoActual().equals(elemento.toString()) ))
return true; }
return false;
}
.....
protected abstract Object elementoActual();
protected abstract Iterador crearIterador();
}
```

Buscar() es un algoritmo implementado en términos de dos métodos abstractos crearIterador() y elementoActual(). Cada estructura concreta redefine los métodos abstractos para personalizar el método de plantilla.

Ambas versiones se encuentran disponibles en línea: *ensamble del jean* y *estructuras abstractas*.

Ventajas:

1. Facilita la encapsulación de funcionalidades comunes a objetos concretos. Cada objeto personaliz su propio comportamiento.
2. Permite que se invierta el paso de mensajes: de la clase madre a la clase hija, lo contrario es lo tradicional.

3. La implementación es bastante flexible. Puede ser con métodos concretos, abstractos, invocaciones a otras clases, etc.

Desventajas:

1. No es fácil distinguir comportamientos comunes que sean susceptibles de ponerse en una clase abstracta. Por lo general se presenta el problema de la trasparencia y la seguridad de los métodos heredados.

Patrón Iterador

Objetivo. Accesar los elementos de un objeto compuesto o agregado sin exponer su representación interna.

Motivación. La navegación en los elementos de una estructura es una tarea frecuente que comúnmente la realiza la estructura misma. La interación puede ser en diferentes sentidos: orden, orden inverso, en paralelo y/o en diferentes etapas. La interfase de las estructuras define, además de las tareas de navegar, operaciones cotidianas: altas, bajas, cambios, etc. La complejidad de la abstracción de la interfase está en función directa de la funcionalidad deseada. Es conveniente, por tanto, diseñar interfaces especializadas en tareas simples y sencillas que incrementen la funcionalidad de los objetos. Así se promueve la comunicación entre los mismos y se disminuye el número de las clases diseñadas.

El patrón Iterador ofrece una solución al separar las tareas que realiza una estructura. Específicamente encapsula las operaciones que permiten navegar por la arquitectura de cualquier estructura.

Analogía con el caso de estudio. La complejidad en las lineas de producción hace casi imposible que una sola persona conozca todo el proceso de confección de un jean. Por lo general, se tiende a especializar las tareas. El patrón Iterador ofrece una solución al permitir que cada ensamble designe quien va a ser la costurera encargada de confeccionar las piezas que lo integran.

Estructura. Se caracteriza por dos clases abstractas: EstructuraAbstracta e Iterador. La primera clase define una interfase común a las estructuras que son implementadas por clases concretas. La segunda clase define una interfase común a los iteradores de cada una de las estructuras. Cada estructura decide cuando crear el iterador que navege por su elementos. A su vez, el iterador decide la forma en que va a recorrer la estructura que lo instanció. El diagrama se observa en la figura 3.14.

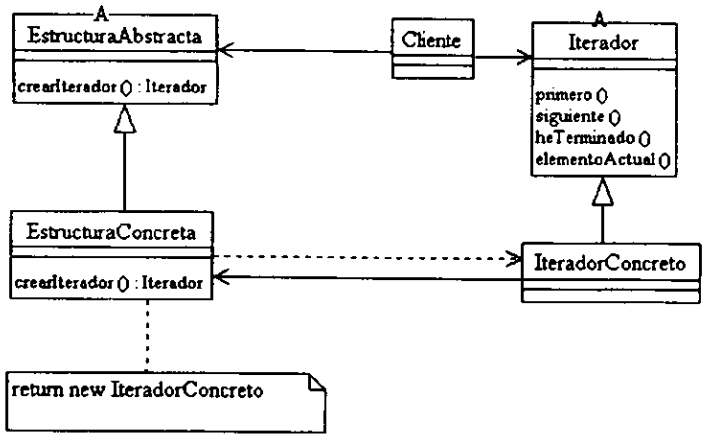

Figura 3.14. Estructura del Patrón Iterador

Variantes para la implementación. Las más relevantes son las siguientes (las que se implementaron se señalan).

1. **Quién controla la interacción.** La puede controlar el cliente o el propio iterador. Si es el cliente, el iterador es llamado externamente, si es el propio iterador, el iterador es llamado internamente. En nuestro caso se desarrolló un iterador interno.
2. **Quién define el algoritmo de navegación.** Se puede definir en la estructura agregada o en el iterador. Si es en la estructura agregada, el iterador guarda sus estados, éste enfoque es llamado cursor. Si es el iterador quien lo define, hay mayor flexibilidad para accesar estructuras con el mismo algoritmo. En el caso de estudio se siguió la segunda opción.
3. **Qué tan robusto es el iterador.** El acceso de los elementos de la estructura agregada no debe interferir con otras operaciones. Un iterador robusto se actualiza cada vez que se efectua una operación en la estructura. La implementación de iteradores robustos se pueden ver en [*31*].
4. **Diferentes iteradores.** Se pueden definir diferentes iteradores y con diferentes privilegios. En el caso de estudio, se prefirió la flexibilidad y sencilles. El iterador es instanciado por su estructura, la cual solo le permite que efectue un número limitado de operaciones sobre sus elementos.

Patrones relacionados. Compositor, Método de Fabricación y Temporal.

Implementación del patrón Iterador. En esta sección se describen dos versiones del patroón, la primera con el caso de estudio, y la segunda con las estructuras abstractas: pila, arreglo y árbol binario.

Ensamble del Jean

El iterador es llamado Costurera, la cual es una clase abstracta. Las clases concretas se especializan en un tipo particular de confección. En el caso de estudio, se especifica que solo se tienen costureras especializadas en la costura de jeans. La estructura sobre la cual se va a navegar es la arquitectura del

jean. Cada ensamble instancia una costurera para que confeccione cada una de las piezas, figura 3.15. La forma en que se comunican los objetos se observa el diagrama de secuencias de la figura 3.16.

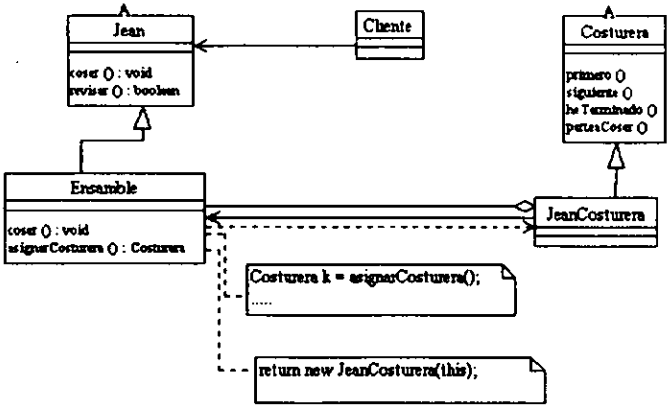

Figura 3.15. Implementacion del Patrón Iterador con el ensamble de un jean. El iterador es JeanCosturera.

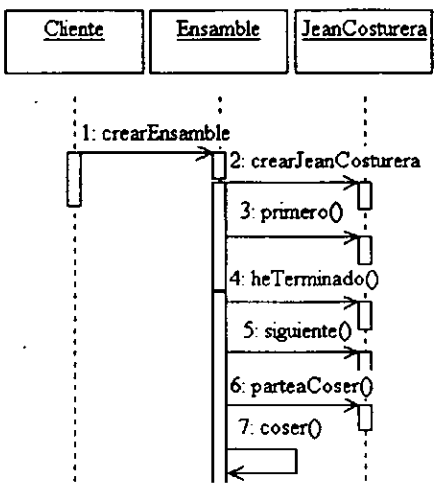

Figura 3.16. Diagrama de secuencias de la implementacion del patrón iterador con el ensamble de un jean. El iterador es JeanCosturera.

Ensamble

```
public class Ensamble extends Jean
{
```

```
private Object [] parte;
...
public void coser()
{
Costurera k = asignarCosturera();
for(k.primero(); // primera pieza
!k.heTerminado(); // no es la última pieza
k.siguiente(); // siguiente pieza
{
((Jean)k.parteaCoser()).coser();
}
}

public Costurera asignarCosturera()
{
return new JeanCosturera(this);
}
...
public boolean revisar(Objet pieza)
{
...
}
...
}
```

El método revisar es similar a coser(). Cada ensamble es revisado por una costurera y cada ensamble determina quien es ella.

El tipo de dato que define a las partes de un ensamble específico es del tipo genérico Object, es por eso que se necesita un "cast" del tipo Jean en coser().

El código se encuentra disponible en línea: *ensamble del jean con su iterador*.

Estructuras Abstractas

En la clase abstracta EstructuraAbstracta, figura 3.17, se define la operación buscar() que es un ejemplo del patron método de plantilla. buscar() invoca la operación abstracta crearIterador() que cada una de las estructuras, arreglo, pila y árbol binario, implementa. Así cada estructura crea su propio iterador. Cada iterador implementa operaciones propias de la estructura que itera. La comunicación entre los objetos se observa en el diagrama de secuencias de la figura 3.18.

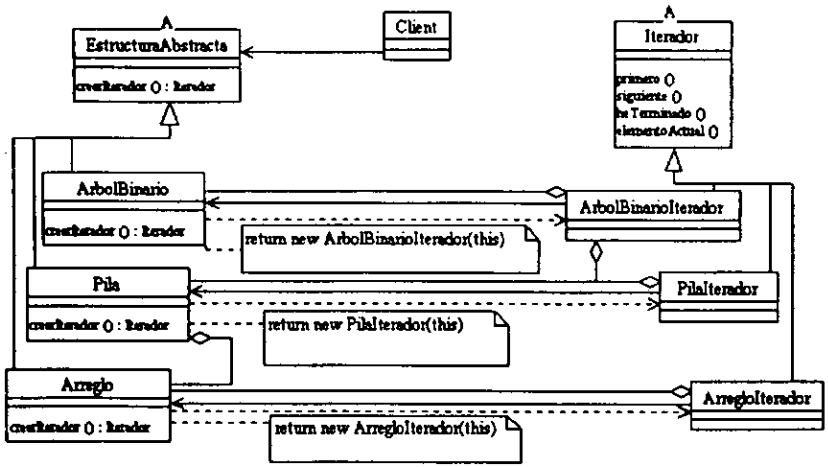

Figura 317. Implementación del patrón Iterador con las estructuras: Arbol Binario, Pila y Arreglo. Cada estructura instancia y configura su iterador. El iterador usa los servicios de la estructura que itera. ArbolBinarioIterador además se auxilia de Pila.

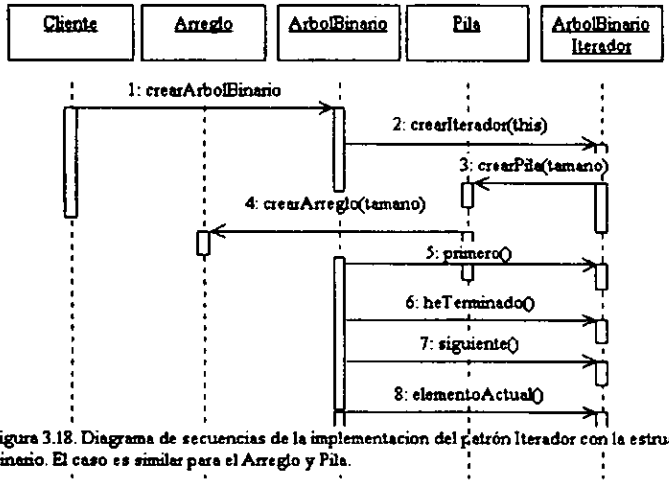

Figura 3.18. Diagrama de secuencias de la implementacion del patrón Iterador con la estructura Arbol Binario. El caso es similar para el Arreglo y Pila.

La pila se implementó con un arreglo, y el iterador del árbol binario con una pila. La pila es la única estructura cuyo iterador no navega por sus elementos; las operaciones: primero(), siguiente() y elementoActual(), se definen de tal forma que siempre regresen el elemento tope.

La clase Arreglo implementa de dos formas diferentes elementoActual(), dependiendo si lo invoca el iterador del arreglo o la pila.

Arreglo

```
public class Arreglo extends EstructuraAbstracta
{
...

public Object elementoActual(int indicex)
{
return arreglo1[indicex];
}
...
public Object elementoActual()
{
return arreglo1[i-1];
}
....
}
```

El método elementoActual(indice) del arreglo, regresa el elemento indicado por el índice y actualizado por primero() y siguiente().

El árbol de busqueda binaria está implementado con referencias de objetos, pero su iterador ArbolBinarioIterador se auxilia de una pila para recorrer sus elementos. La implementación lograda se basa en una idéa desarrollada por T. Budd [*32*]. El procedimiento para navegar en el árbol es como sigue: Un árbol se puede recorrer en cualquiera de tres formas, inorden, posorden y preorden. Una vez usando cualquiera de estas formas, es posible encontrar un elemento en particular pero tendríamos que modificar la interfase del árbol para identificar cada elemento. La situación se complica porque el algoritmo es recursivo y no hay referencias explicitas de cada nodo a su padre. Para simplificar el proceso, únicamente se implementó el recorrido en inorden. Conforme se va recorriendo el árbol, se almacenan en la pila los nodos izquierdos hasta llegar al elemento más a la izquierda, que es el tope de la pila y el primer elemento en la iteración. El segundo elemento es el ancestro del primero, para ello se saca el tope de la pila y se recorre la rama derecha del tope, si es vacía entonces se tiene el segundo elemento. Si no, hantes de perder la referencia, se recorre su rama derecha en inorden y se almacenan los nodos izquierdos; nuevamente el elemento siguiente es el tope, se saca y se hace el mismo análisis.

Arbol Binario

```
public class ArbolBinarioIterador extends Iterador
{
private ArbolBinario raiz;
private Pila arbolPila;
...
public void primero()
{
arbolPila.inicializarPila();
inorderAuxiliar(raiz);
}

public void inorderAuxiliar(ArbolBinario nodoActual)
{
```

```
while(nodoActual != null)
{
arbolPila.agregar(nodoActual);
nodoActual = nodoActual.darArbolIzquierdo(); // unicamente se almacenan los nodos izquierdos;
}
}
...
}
```

El método elementoActual() muestra el contenido del tope de la pila.

El siguiente elemento se encuentra de la siguiente forma: pop() regresa el tope de la pila que es un nodo, si su nodo derecho es nulo entonces el tope de la pila es el siguiente, si no se invoca nuevamente a inorderAuxiliar(siguiente) que metera a la pila los nodos izquierdos del nodo derecho actual. El proceso es complicado, más aún si se realiza con tipo de datos genéricos como en nuestro caso.

```
public void siguiente
{
ArbolBinario nodo, siguiente;
if(!arbolPila.pilaVacio())
{
nodo=(ArbolBinario)arbolPila.borrarElemento(); // pop()
siguiente=nodo.darArbolDerecho();
if(siguiente!=null)
inorderAuxiliar(siguiente);
}
}
```

El código se encuentra disponible en línea: *estructuras abstractas con su iterador*.

Ventajas:

1. Permite recorrer estructuras agregadas de diferentes formas. Es factible definir diferentes iteradores, cada uno con su propio algoritmo. La estructura puede instanciar el iterador deseado.
2. Simplifican la interfase de la estructura agregada, al separar las operaciones en las que se especializa la estructura de las que no; simplifica su interfase.
3. Iteradores en paralelo. Cada instancia del iterador define su algoritmo, por lo que se pueden tener varios iteradores al mismo tiempo.

Desventajas:

1. Diversos métodos implementados por los iteradores concretos no tienen sentido en las estructuras, tal es el caso de la pila. La solución adoptada no permite recorrer la pila, porque las únicas operaciones definidas son push() y pop(). El iterador es dependiente de los métodos definidos en la estructura que itera.

Patrón Visitador

Objetivo: Agregar una nueva responsabilidad a los elementos de una estructura de objetos sin cambiar su clase.

Motivación: Consideremos una estructura jerárquica similar a la que un compilador utiliza para representar una gramática. Los nodos del árbol tienen objetivos específicos, además de que operaciones como la sintaxis estática, optimización, y asignación de variables, son excluyentes por nodo, es decir, la operaciones tiene diferente significado en los nodos que representan tareas de asignación, variables de acceso, expresiones aritméticas. Cada nodo corresponde a una clase y su definición es dependiente del lenguaje que se esta compilando. La definición del compilador promueve que la arquitectura en algunas de sus fases siga una estructura jerárquica, sin embargo no todas las operaciones se aplican igual en todos los nodos. Es deseable que cada nodo se especialize en operaciones que definan su comportamiento y no dejen de implementar otras que no tienen sentido.

El patrón Visitador ofrece una solución al definir operaciones que representan una nueva responsabilidad para la clase que visita. Por otra parte, la clase que es visitada define en su interfase el protocolo que le permite tener visitantes.

Analogía con el caso de estudio. La estructura jerárquica del jean esta representada por elementos simples y compuestos. Las operaciones que se efectuan en cada una de las piezas es diferente, siendo en unas más complejas que en otras. En este caso, la operación que no forma parte de la interfase del jean, ni de ninguno de sus elementos que componen la jerarquía es la de inspección de la confección o contro de calidad. El control de calidad no tiene que formar parte de las operaciones propias del jean, sino de un agente especializado.

Estructura: La estructura está representada por dos jerarquías de clases, la que visita y la que es visitada: Visitador y Elemento, respectivamente. Elemento define un método que permite que él mismo (this) se pase como parámetro a Visitador, que es donde se implementa la funcionalidad deseada. Ver figura 3.19.

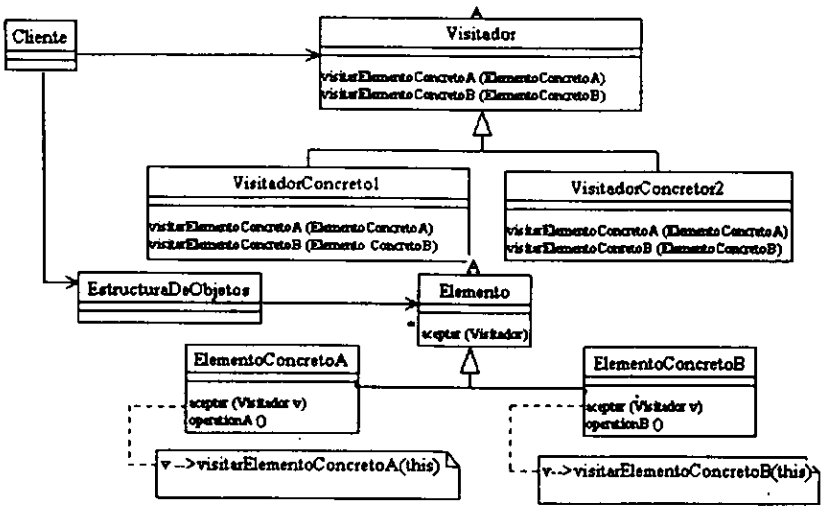

Figura 3.19 . Estructura del patrón Visitador.

Variantes para la implementación: Las más relevantes son las siguientes (las que se implementaron se señalan).

1. **"Double Dispatch:"** La técnica en la que se basa Visitador es conocida como "double dispatch"; la operación que añade la nueva funcionalidad depende de las dos estructuras, la que permite que la visiten y la que visita. El método aceptar() es "double dispatch".
2. **Quien es reponsable de la navegación por la estructura.** Puede ser implementada con el patrón iterador, o bien dejar que la misma estructura recorra sus elementos. En el caso de estudio, la estructura corresponde al patrón compositor, lo que permite recorrer los elementos en forma recursiva. Se prefirió éste enfoque para simplificar la descripción del patrón Visitador.

Patrones relacionados: Compositor, Interpretador e Iterador.

Implementación del patrón Visitador. La operación adicional y anexada a la interfase del ensamble del jean es la de control de calidad. La clase ControlDeCalidad regista el número de piezas defectuosas por ensamble. El jean define en su interfase una operación abstracta, que cada estilo de jean implementa y que permite tener visitantes. Visitador es una clase abstracta que capta el protocolo para visitar la estructura de jean. Visitadores concretos representan tareas específicas que no forman parte de jean pero que es necesario realizarlas, por ejemplo, control de calidad, control de inventario, y pedidos. En las figuras 3.20 y 3.21 se observa la estructura de la implementación y la comunicación entre los objetos participantes, respectivamente.

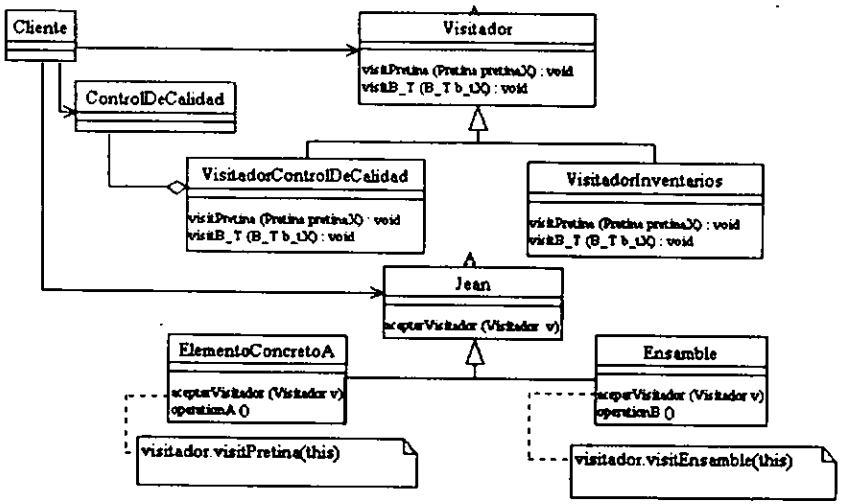

Figura 3.20. Implementación del patrón Visitador con el ensamble de un jean, control de calidad e inventarios.

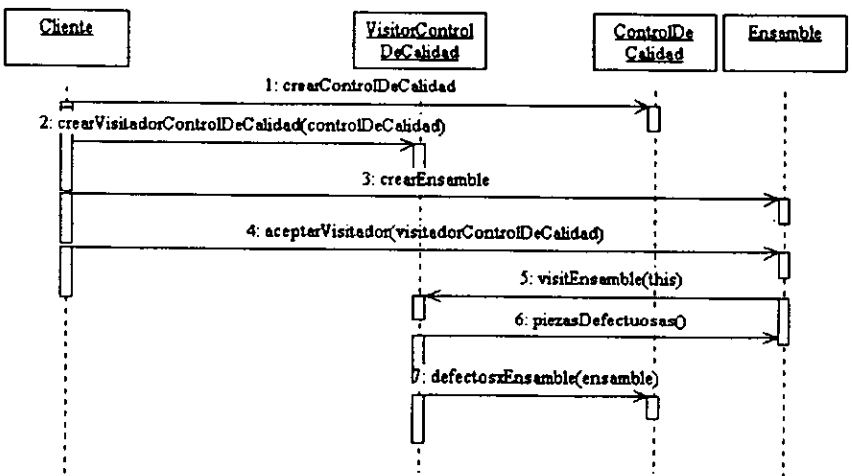

Figura 3.21. Diagrama de Secuencias de la implementación del patrón Visitador con el ensamble de un jean y control de calidad.

Jean

```
public abstract class Jean
{
...
public void coser()
```

```
{
cosido = true;
piezasDefectuosas = (int)(java.lang.Math.round(java.lang.Math.random()*100));
}
...
public abstract void acceptVisitor(Visitador visitador);
}
```

Unicamente se obtienen piezas defectuosas por ensamble.

La clase Visitador define operaciones abstractas que permitan visitar a los elementos de jean. Se define una operación visitador por cada elemento de la estructura. El parámetro de cada operación es el elemento que desea la nueva funcionalidad. El visitador concreto efectua o invoca una operación sobre su parámetro.

Visitador

```
public abstract class Visitador
{
...
protected abstract void visitPretina(Pretina pretinaX);
protected abstract void visitB_T(B_T b_tX);
....
protected abstract void visitEnsamble(Ensamble ensambleX);
}
```

Control de Calidad

```
class VisitadorControlDeCalidad extends Visitador
{
private ControlDeCalidad controlDeCalidad;
...
public void visitPretina(Pretina pretinaX)
{
....
}
// El cuerpo de cada método no se implementa, salvo el de los ensambles

public void visitEnsamble(Ensamble ensambleX)
{
controlDeCalidad.defectosxEnsamble(ensambleX.miNombreEs(), ensambleX.piezasDefectuosas());
}
...
}
```

VisitadorControlDeCalidad invoca a la clase especializada en el control de calidad ControlDeCalidad.

En forma semejante, se pueden definir diferentes funcionalidades. La estructura que representa la clase abstracta Jean no sabe como realizar el control de calidad, es por ello preferible que una clase especializada en esas tareas las realize.

El código de la implementación se encuentra disponible en línea: *el control de calidad en el ensamble del jean.*

Ventajas.

1. Permite definir operaciones alternas a las ya establecidas en las interfaces de una estructura jerárquica. Cada visitador define una nueva funcionalidad.
2. Un Visitador reune operaciones comúnes y separa las que no son. Cada subclase se aboca a un conjunto de ellas.

Desventajas.

1. Es dificil mantener la misma interfase de la estructura, sobre todo si se agrega o elimina un elemento. Como consecuencia, la interfase de Visitador también se tiene que cambiar.
2. Visitador puede visitar diferentes estructuras y con diferentes padres. Si la navegación se implementa con el patrón Iterador, Visitador limita su flexibilidad ya que Iterador itera sobre jerarquías con un mismo padre.
3. Dependiendo de la implementación, Visitador puede o no romper el encapsulamiento.

Capítulo 4

Capítulo 4. Conclusiones

El enfoque del presente trabajo difiere en varios aspectos del catálogo de patrones de diseño de Gamma et al [3], texto que se tomó como base. El Catálogo es una recopilación de diseños extraídos principalmente de frameworks, *ver glosario*. Un framework es un diseño global basado en otros más simples. Un framework y un patrón se distinguen básicamente por los siguientes tres aspectos:

- **Los patrones de diseño son más abstractos.** Los frameworks representan diseños listos para ser utilizados, en contraste, pocos son los patrones que directamente se pueden usar.
- **La arquitectura de los patrones de diseño es más específica.** Un framework comúnmente contiene diversos patrones, lo contrario no es cierto.
- **El dominio de los patrones de diseño es más amplio.** El diseño que representa el patrón es independiente del dominio. El usuario es libre de modificar el diseño e implementarlo en su propio contexto de estudio. Un framework es más rígido, y limitado; su arquitectura no se puede cambiar y se diseñan lo más ampliamente posible para que puedan ser utilizados por cualquier usuario.

Uno de los objetivos que persigue el catálogo de patrones es trasmitir el conocimiento de diseñadores expertos a pioneros en varios dominios. Diversos aspectos motivan que el objetivo no se cumpla, entre los cuales se encuentra la extracción directa que se realizó de los frameworks y la forma de presentarlos. La inquietud por transmitir el conocimiento encapsulado en el patrón e incrementar su audiencia, ha sido uno de los pilares durante el análisis, diseño e implementación de los patrones en este trabajo.

La base medular es el catálogo de patrones de diseño, él cual es, sin duda, el que ha abierto la pauta para el estudio de una nueva área de la ingeniería de software: *ingeniería de software orientada a patrones de objetos*. Las ideas ahí expuestas se retomaron y se les dió un matiz diferente. Los puntos más sobresalientes son los siguientes:

- **Primera parte:**
  - Se hizo una breve investigación sobre el origen de los patrones en la arquitectura, y su posterior incorporación en la ingeniería de software.
  - Se analizaron diversas definiciones y se elaboró una propia.
  - Se elaboró una clasificación de los patrones por fases que corresponden a las diferentes etapas del desarrollo de software: análisis, diseño e implementación. La clasificación se basó en la identificación de los patrones de tres autores y sus colaboradores: P. Coad et al., patrones de análisis [*21*], E. Gamma et al., patrones de diseño [*3*], y J. Coplien, patrones de código [*34*]. Los patrones de análisis y de diseño siguen una organización con varios criterios de exposición; los de código están más dispersos, por lo que su comprensión es limitada. No obstante que la clasificación propuesta es bastante útil, no existe aún una por disciplina.
  - La terminología de los patrones de diseño tiene varios orígenes. Su gestación visitó varios países, además de que más de la mitad de ellos fueron originalmente escritos en alemán. El nombre que distingue uno de otro, muchas veces no corresponde al diseño que representa, es por ello que se decidió renombrarlos con base en su contexto.
  - En la última parte de ésta sección, se describieron varios consejos prácticos para seleccionar y usar un patrón de análisis, de diseño y de código, en las diferentes etapas del desarrollo del software. El catálogo presenta una similar, pero es exclusivamente de los patrones de diseño.

De los patrones de código muy poco se ha dicho, de hecho sólo existe documentación de los encontrados en los lenguajes Smalltalk y C++.

- **Segunda parte:**
  - El objetivo de ésta sección fué el análisis, diseño e implementación, utilizando las técnicas tradicionales del modelo orientado a objetos, de un caso práctico "el ensamble de un jean". El ejemplo es posteriormente re-estructurado, en la tercera parte, con el enfoque de patrones de diseño.
  - En la etapa del análisis, se empleó la reingeniería de procesos y la ingeniería de software orientada a objetos de I. Jacobson et al. [*14,10*], para definir el dominio del problema, la interacción que tienen los actores del sistema con el mismo, la descripción del proceso que genera la solicitud de un servicio al sistema "caso de uso", y la interacción entre los objetos de un caso de uso "diagrama de interacciones". Unicamente se describió el caso de uso "coser un jean". Posteriormente se empleó la técnica de las tarjetas CRC [*24*], para identificar las responsabilidades de cada clase y sus colaboradores.
  - En la etapa del diseño, se empleó la metodología de G. Booch [*4.5*], para identificar las relaciones e interacciones entre las clases. Los diagramas de clases se graficaron con el lenguaje de modelado UML [*4*], por ser el más robusto hasta la fecha. UML es un lenguaje más abstracto que sus antecesores, sin embargo, tiene ciertas peculiaridades que pueden dificultar la comunicación entre los diseñadores. Por ejemplo, la relación de dependencia ---->, se puede interpretar como "usar" en el sentido en que lo maneja Booch, o instanciación cuando una clase crea una instancia de otra clase. Para evitar ambivalencias y ser más preciso en los términos, fué necesario agregar una letra A para simbolizar una clase abstracta en lugar de denotarla (abstract) como lo hace UML.
  - En la etapa de la implementación, se utilizó el lenguaje java. Los extractos de código que aparecen en el catálogo de patrones de diseño corresponden a C++. Java es simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutral, portable, y dinámico, entre otras cualidades, además de que se esta volviendo un lenguaje bastante popular.
- **Tercera parte:**
  - El objetivo de esta sección fué la identificación, e implementación de los patrones de diseño para re-estructurar el caso de estudio práctico. Primero se buscó la arquitectura adecuada que diera soporte al ensamble del jean con los patrones de estructura, el patrón *compositor* fué el más indicado (compone y/o ensambla objetos en estructuras de árbol).
  - Enseguida se buscó la forma de crear los diferentes tipos de objetos, primitivos y compuestos, con los patrones de creación. Se adecuaron tres: *Método de Fabricación. Fábrica Abstracta y Constructor*. Los objetos hoja y/o primitivos se crearon con *Método de Fabricación*; las familias de jeans, clásico, stretch y acampanado, con *Fábrica Abstracta*; los ensambles con *Constructor*. Rara vez un patrón existe en forma aislada, por lo general el buen desempeño de uno depende de otro, o bien, uno es un caso particular o general de otro (s). En nuestro caso, *Constructor* se implementa con *Fábrica Abstracta* y *Fábrica Abstracta* con *Método de Fabricación*.
  - La comunicación e iteración entre los objetos se realizó con los patrones de desempeño: *Método de Plantilla*, *Iterador* y *Visitador*. *Método de Plantilla* permitió definir un algoritmo para asignar costureras especializadas en las diferentes etapas del proceso de confección de un jean. *Iterador* ayudó a diseñar varias formas en las que fuera posible navegar por la estructura jerárquica del jean. *Visitador* se empleó para añadir funcionalidades a los objetos en tiempo de ejecución. En el caso del patrón *Iterador*, se implementó un segundo ejemplo, con tres estructuras abstractas y sus respectivos iteradores: un arreglo, un árbol binario y una

pila, aunque no se implementaron los 24 patrones del catálogo, los seleccionados son los más representativos.

- **Notación Gráfica Utilizada:**
  - El lenguaje de modelado (UML) es el sucesor de los métodos de análisis y diseño orientados a objetos que aparecieron a finales de la década de los 80's y principios de los 90's. Unifica el método de Booch, Rumbaugh y Jacobson. A la fecha, UML está en un proceso de estandarización con OMG (Object Management Group).
  - UML es un lenguaje de modelación, no un método. La mayoría de los métodos consisten de un lenguaje de modelado y un proceso. El *lenguaje de modelado* es la notación gráfica que los métodos usan para expresar diseños. El *proceso* son los pasos que se tienen que seguir en la elaboración del diseño.
  - UML no incluye los diagramas de la reingeniería de procesos de Jacobson, los cuales se utilizaron en el caso de estudio, figuras 2.1 y 2.2. El diagrama de Gantt, figura 2.6, es un concepto de descomposición funcional tomado de la administración de organizaciones.
  - La agregación <> se utiliza en el sentido en que Booch lo define, la agregación es una relación en la cual la *parte* se integra para formar el *todo*, pero la *parte* vive y muere independientemente del *todo*.
  - La herencia, a diferencia de Booch que es una relación *es un (a)* para generalizar, se utiliza en el sentido de clasificar. Así evitamos incongruencias como decir que el elemento *pretina* es un *jean*.
  - La dependencia ----> se emplea para denotar la relación *usando* de Booch y la instanciación de objetos.
  - En la versión 4.0 de UML no se incluye un símbolo para las clases abstractas, por ello fue necesario agregar la letra A en la parte superior de las clases.

El presente trabajo es un esfuerzo conjunto de su autor y asesora para la divulgación de la ingeniería de software orientada a patrones de objetos.

Ciudad Universitaria, 19 de enero de 1998.

Bibliografía

1. Adele Goldberg and Kenneth S. Rubin, Succeding with Objects, Addison Wesley, 1945.
2. Martin Fowler, Analysis Patterns: Reusable Object Models, Addison Wesley, 1977.
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object Oriented Software, Addisson Wesley Pu. Co. 1995.
4. Martin Fowler, Kendall Scott, UML Distilled, Applying the Standard Object Modeling Language. Addison Wesley, 1997.
5. http://www.rational.com/ot/uml/technical_papers/index.html.
6. Jalai Feghhi, java beans, Coriolis Group Books, 1997.
7. Robert Orfall, Dan Harkey, Client Server Survival Guide, Van Nostrand Reinhold Pu. Co., 1994.
8. Berezuk, S., Finding solutions through pattern languages. IEEE Computer 27, 12. 1994.
9. Grady Booch, Object Oriented Analysis and Design with Applications, second edition, Benjamin Cummings Pu. Co. Inc. 1994.
10. Ivar Jacobson, Object Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley, 1994.
11. André Weinand, Erich Gamma, and Rudolf Marty. ET++ An Object Oriented Application Framework in C++. In Object Oriented Programming System, Languajes, and Applications Conference Proceedings, S.D.C.A. September 1988, ACM Press.
12. John M. Vlissides and Mark A. Linton. Applying Object Oriented Design to Structured Graphics. In Proceeding of the 1988 USENIX C++ Conference, page 81-941, Denver, Co., Octuber 1988. USENIX Association.
13. Paul R. Colder and Mark A. Linton Glyphs: Flyweight object for user interfaces. In ACM User Interface Software Technologies Conference, pages 92-101. Snowbird, UT. October 1990.
14. Ivar Jacobson, Maria Ericsson y Agneta Jacobson, The Object Advantage, Business Process Reenginering with Object Technology, Addison Wesley Pu. Co., 1994.
15. Luca Cardelly, Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism, Computing Surveys, Col. 17, No. 4, Dec. 1985.
16. Alexander C, The Timeless Way of Building, Oxford University Press, 1979.
17. Alexander C., Ishikawa S. y Y. Silverstein, A Pattern Languaje, Oxford University Press, 1977.
18. Alexander C., The Oregon Experiment, Oxford Univesity Press, 1975.
19. Alexander C., Notes on the Synthesis of Form, Harvard University Press, 1964.
20. Buschmann F. y Meunier R., A System of Patterns, John Wiley and Sons, 1996.
21. Coad P., North P., y Maffield M., Objects Models Strategies, Patterns and Applications, Yourdon Press, P.H. 1995.
22. Brummond N., Object Oriented Analysis and Design using CRC Cards. http://wwws.csc.calpoly.edu/~dbutlerg/tutorials/winter96/crc_b/overview.html.
23. Patrick Niemeyer, Joshua Peck, Exploring Java, O'Reilly & Associates, Inc., 1996.
24. James Gosling, Bill Joy, Guy Steele, The Java Languaje Specification. Addison Wesley, 1996.
25. Fayad M.E., Schmidtt, D.C. Johnson P.E., Object Oriented Application Frameworks: Implementation and Experience, John Wiley and Sons., N.Y. 1977.
26. Fayad M.E., Schmidt D.C., Hohnson R.E., Object Oriented Applications Frameworks: Problems and Perspectives, John Wiley and Sons. N.Y., 1977.
27. Communications of the ACM, Volume 40, Number 10, Object Oriented Application Frameworks.
28. R. Milner, "A Theory of Type Polymorphism in Programming", Journal of Computer and System Sciences, Vol. 17, pp. 348-375, 1978.
29. Bertrand Meyer, Object Oriented Software Construction, Series in Computer Science, P. H. N.T.

1988.
30. John M. Vlissides, Mark A. Linton, Unidraw: A framework for building domain specific graphical editors. ACM Transations on Information Systems, 1990.
31. Tomas Kofler, Robust Iterators in ET ++. Structured Programming, March 1993.
32. Timothy A. Budd, Classica Data Structures in C++. Addison Wesley Pu. Co. 1994.
33. http://hilside.net/patterns/patterns.html.
34. James O. Coplien, Advanced C++ programming Styles and Idioms, Addison Wesley, 1994.
35. Erich Gamma, Object Oriented Software Development based on ET++; Design Patterns, Class Library, Tools, in German. Ph D thesis, University of Zurich. Institud fur Informatik, 1991.
36. Erich Gamma, Object Oriented Software Development based on ET++. Design, Class Library, Tools, in German. Springer Verlag, Berlin, 1992.
37. William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in design application framework and evolving object oriented systems. In SOOPPA Conference Proceedings, p. 145-161. Morist College, N.Y., ACM Press, September 1990.
38. Alan Snyder. Encapsulation and inheritance in object oriented languajes. In Object Oriented Programming Systems, Languajes, and Applications Conference Proceedigns, p. 38-45. Portland, Or. ACM Press, November 1986.
39. Bertrand Meyer, Reusable Software. The Base Object Oriented Component Libraries. Prentice Hall, 1994.
40. Davie C. Hay. Data Model Patterns Conventions of Thought. Darset House Pu. Co. 1995.
41. Robert Orfali, Dan Harkey, Jery Edwards. The Essential Distributed Objects Survival Guide, John Wiley & Sons, 1996.
42. http://www.rational.com/demos/rose4demo/.
43. Theory and Practice of Object Systems, Volume 2, number 1, 1996.
44. J. O. Coplien. The History of Patterns: http://c2.com/cgi/wiki?HistoryOfPatterns.
45. D.C. Schmidt, A Systems of Reusable Design Patterns for Communication Software, Theory and Practice of Object System, Specific Issue on Patterns and Pattern Languajes, S.P. Berczok. John Wiley and Sons, 1995.
46. Alan Snyder, Encapsulation and inheritance in object oriented languajes. In Object Oriented Programming Systems, Languajes and Applications conference Proceedings, p. 38-45, Portland, OR. ACM PRESS, November 1986.
47. H.G. Baker, List Processing in Real Time on a Serial Computer. A.I. Working Paper 139. MIT-AI Lab. Boston EEUU. April 1997.
48. Jourdon E. and Constantine L., Structured Design: Fundamentals of Discipline of Computer Program and System Design, englewood clifs, N. Y. Prentice Hall.
49. Yourdon E. Modern Structures Analysis. Yourdon Press, Prentice Hall.
50. Ross D. Applications and extensions of SADT, IEEE Computer, April 1985.
51. R. Johnson, V. Russo, Reusing Object Oriented Designs, University of Illinois, Paper, 1995.
52. Bjarne Stroustrup, The Design and Evolution of C++, Addison Wesley, Reading, MA. 1994.

Glosario

1. **Abstracción.** Denota las características esenciales de un objeto que lo distingue de otros objetos y limita la conceptualización que un observador tiene de él.
2. **Actor.** Es un elemento en el modelo de los casos de uso que representa un role.
3. **Análisis orientado a objetos.** Es un método de análisis que examina los requerimientos desde la perspectiva de clases y objetos encontrados en el dominio del problema.
4. **Atributo.** Una característica, adjetivo, que clasifica un objeto y que forma parte de su definición semántica en el modelo de objetos.
5. **Caso de uso.** Es una secuencia de transacciones en un sistema, cuyo objetivo principal es aportar un valor medible al actor del sistema.
6. **Clase.** Es un concepto lógico para describir un conjunto de objetos que comparten una estructura y un comportamiento. Los términos "tipo" y "clase" frecuentemente se usan en forma intercambiable; una clase encapsula la estructura y el comportamiento de un conjunto de objetos, un tipo se refiere a la interfase de una clase. Una clase puede tener varios tipos.
7. **Clase abstracta.** Una clase que no tiene instancias. Una clase abstracta capta la estructura y el comportamiento común de las subclases. Las subclases implementan los métodos declarados abstractos.
8. **Clase concreta.** Una clase cuya implementación está completa y por tanto, puede tener instancias
9. **Constructor.** Una operación de la clase que crea y inicializa el estado de un objeto.
10. **Diagrama de interacciones.** Modelos que describen la colaboración en el desempeño de un grupo de objetos de un sólo caso de uso. Se dividen en diagramas de secuencia y de colaboración. La diferencia entre ambos diagramas es el formato para representar la interación entre los objetos.
11. **Diseño orientado a objetos.** Método de diseño que incluye el proceso de la descomposición orientada a objetos y una notación para describir el modelo bajo diseño en sus diferentes formas: lógica, física, estática y dinámica.
12. **Encapsulación.** Es el proceso de ocultar la abstracción (atributos e implementación de los métodos) de un objeto. La representación (atributos) de un objeto no es visible al mundo exterior. Las operaciones son el único medio por el cual se puede accesar y modificar la representación de un objeto.
13. **Framework.** Es un conjunto de clases que cooperan para hacer un diseño reusable de un software específico.
14. **Interfase.** Es el conjunto de todas las operaciones de una clase, en donde una operación únicamente se refiere a la "signature" de un método: visibilidad, tipo, nombre y parámetros. La interfase describe el tipo de respuestas que un objeto puede ofrecer.
15. **Ligado dinámico.** Asociación en tiempo de ejecución de la petición de un servicio y el método de objeto que satisface dicha solicitud.
16. **Monomorfismo.** Un concepto en la teoría de tipos, en el cual un nombre, como la declaración de una variable, sólo puede denotar objetos de la misma clase.
17. **Objeto.** Es una abstracción caracterizada por un estado, comportamiento e identidad. Los términos instancia y objeto son comúnmente intercambiables.
18. **Objeto distribuido.** Es un componente y/o objeto en un ambiente abierto de cliente-servidor con propiedades y caracterisitcas que le permiten ser distribuido.
19. **Operación.** Se refiere a la descripción de una responsabilidad en una clase. La operación denota que es lo que hace la responsabilidad y el método cómo lo hace. El conjunto de operaciones definen la interfase de la clase.

20. **Patrón.** Es la solución a un problema recurrente en un contexto particular.
21. **Persistencia.** La propiedad que tiene un objeto para hacerse trascendente en el tiempo.
22. **Polimorfismo.** Un concepto en la teoría de tipos, en el cual un nombre, como la declaración de una variable, puede denotar objetos de diferentes clases relacionados por una clase padre o superclase.
23. **Privado.** Una declaración que forma parte de la interfase de una clase, objeto o módulo. Lo que es declarado como privado no es visible a otras clases, objetos o módulos.
24. **Programación orientada a objetos.** Es un método de implementación en el cual los programas son organizados en base a la colaboración de un conjunto de objetos, cada uno de los cuales representa una instancia de una clase.
25. **Protegido.** Una declaración que forma parte de la interfase de una clase, objeto o módulo. Lo que es declarado como protegido no es visible a otras clases, objetos o módulos, excepto a las subclases.
26. **Público.** Una declaración que forma parte de la interfase de una clase, objeto o módulo. Lo que es declarado como público es visible a otras clases, objetos o módulos.
27. **Relación.** Elemento de diseño del modelo de objetos para describir semanticamente la interacción entre clases. Frecuentemente se utilizan cuatro relaciones: herencia, asociación, uso y agregación.
    - La herencia es el proceso mediante el cual una clase comparte la estructura y el comportamiento definidos en otra clase (herencia simple) u otras clases (herencia múltiple).
    - La asociación denota una dependencia semántica entre dos clases; si no se establece explícitamente la dirección, se asume que es bidireccional.
    - Usa denota una relación entre dos clases, en la cual una utiliza los servicios de otra.
    - La agregación describe una contención de una clase en otra.
28. **Tarjetas CRC.** Herramienta de análisis que describen las responsabilidades de una clase y las clases con las que colabora para satisfacer ésas responsabilidades.
29. **Tipo.** Es un nombre utilizado para describir una interfase.
30. **UML.** Lenguaje para modelar el análisis y diseño orientado a objetos.