UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONALES Y DE POSGRADO
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

MODELO REUTILIZABLE ORIENTADO A OBJETOS DE EMPRESAS

TESIS

Que para obtener el grado de
MAESTRO EN CIENCIAS DE LA COMPUTACION
p r e s e n t a:

ARI SCHOENFELD LIBERMAN
schoenfeld @ acm.org

Director: DRA. HANNA OKTABA

México, D. F. Agosto de 1998

A mi madre,

Artista, Madre y Corazón

A mis maestros

a quienes nunca podré

regresar lo que me dieron

CONTENIDO

Introducción 5

1. Patrones 9

Definición 9

Historia 9

Formato 10

Clasificación 11

2. El modelo orientado a objetos de empresas 13

Tres patrones de dominio del problema 13

2.1 Personas y Roles 14

Intención 14

Motivación 14

Aplicabilidad 14

Estructura 15

Participantes 16

Colaboraciones 16

Consecuencias 16

Implementación 16

Usos conocidos 17

Patrones relacionados 18

2.2 Documentos y Roles de Personas 18

Intención 18

Motivación 18

Aplicabilidad 19

Estructura 19

Participantes 20

Colaboraciones 20

Consecuencias 20

Implementación 21

Usos conocidos 21

Patrones relacionados 22
2.3 Valores Convertibles 22
Intención 22
Motivación 22
Aplicabilidad 22
Estructura 23
Participantes 23
Colaboraciones 24
Consecuencias 24
Implementación 24
Usos conocidos 25
Patrones relacionados 26
Conclusiones 27
Referencias 28
Notación 30
Anexo del artículo publicado en PLoP '96 31

Introducción

El desarrollo de sistemas es complejo. El desarrollo de sistemas que cumplan con la funcionalidad especificada y que se desarrollen bajo lo presupuestado en tiempo y en dinero, es aún más complejo. La "Crisis del Software" no ha terminado. Se siguen escuchando historias de monumentales fracasos de desarrollos de software [CACM97-40,4] a pesar de todos los avances que ha habido en metodologías, organización de equipos de desarrollo, lenguajes, etc.

El paradigma de la programación orientada a objetos[1] no es nuevo. Para algunos es la panacea que da solución a todos los problemas de desarrollo, y para otros, no aporta ninguna ventaja ante la programación estructurada. Más que perderse en una discusión sofista sin sentido, es importante encontrar las ventajas objetivas que tiene este paradigma.

Simula 67, el primer lenguaje orientado a objetos que en 1967 se terminó de diseñar [Nygaard81] (antes inclusive de que surgiera la programación estructurada) tenía, entre otros, el objetivo de ser utilizado para simulaciones. La idea principal era poder realizar modelos del dominio del problema a simular, los cuales pudieran ser modificados fácilmente al ir comprendiendo más el dominio del problema, o simplemente al querer aumentar complejidad que deliberadamente en un principio se decidió omitir. Para esto, la programación orientada a objetos ofrece algunos mecanismos como la herencia, el polimorfismo, la sobrecarga de operadores, etc. los cuales permiten que el sistema evolucione incrementalmente reutilizando el código ya escrito.

A pesar de que estas ideas ya estaban bastante maduras desde 1967, la programación orientada a objetos apenas hace 10 años comenzó a tener un auge en los círculos académicos y comerciales (como referencia, la revista "Journal of Object Oriented Programming" (JOOP) tuvo su primer ejemplar en marzo de 1988). ¿Porque surgió el interés en este paradigma? Antes que nada, por la crisis de software no superada, y en segundo lugar probablemente por las promesas mencionadas anteriormente: reutilización de código (con todas las ventajas que esto ofrece [Meyer97 pp. 68]) y facilidad de modificación y evolución del sistema. Sin embargo si un lenguaje orientado a objetos no es utilizado para modelar, y sus estructuras y relaciones no son utilizadas para lo que fueron diseñadas, difícilmente se podrán obtener los resultados esperados.

Nadie duda de la popularidad de este paradigma. C++ y Java son lenguajes en boga y aunque no siempre son utilizados correctamente, permiten, promueven e inclusive obligan la entrada a este paradigma de muchos programadores día a día. También es cierto que existen ya muchos sistemas basados en esta programación que funcionan, y hay una tendencia generalizada a realizar casi cualquier sistema bajo este paradigma. A pesar de ello, hacer un sistema completamente orientado a objetos, no es garantía para no fracasar.

¿Cuáles son los aspectos que provocan que un proyecto triunfe y otro fracase? Existen factores psicológicos, sociales, económicos y tecnológicos que lo determinan y sería pretencioso en este texto definir dichos aspectos. Sin embargo, un elemento que define en gran medida todos esos aspectos, es la experiencia. Un programador con experiencia, logra mucho mejor sus objetivos (en funcionalidad, tiempo y costo), que un programador principiante. Por lo tanto, el factor principal para fracasar, es la falta de experiencia y conocimiento en al menos alguno de los factores mencionados. Adicionalmente, la programación orientada a objetos tiene la desventaja

---

[1] Nota: se asume que el lector tiene conocimientos básicos de programación orientada a objetos y que le son familiares conceptos como clase, objeto, herencia, polimorfismo, etc. Como referencia, se recomienda al lector el libro de B. Meyer [Meyer97].

de ser nueva para la mayoría de los programadores activos, y además su curva de aprendizaje es lenta.

Algunas de las dificultades que hacen lento el aprendizaje de este tipo de programación son:

- Cambio de paradigma – es necesario pensar diferente que con otros paradigmas en el diseño e implementación de los sistemas.
- Complejidad – Son muchos los elementos que deben aprenderse para poder comenzar, y muchas veces es necesario entender varios conceptos nuevos para poder utilizar uno de interés (por ejemplo, encapsulación, polimorfismo, herencia y asociación dinámica o dynamic binding. Si no se domina la herencia, y se entiende la asociación dinámica, no es posible utilizar y entender el polimorfismo).
- Incompatibilidad con los conocimientos anteriores – Es inclusive un problema serio el mezclar conceptos de la programación funcional con este paradigma, y los resultados pueden ser catastróficos.
- Conocimientos o conceptos equivocados – Como resultado de la "moda" de la programación orientrada a objetos es común que se aprendan informal y equivocadamente los conceptos de herencia, polimorfismo, etc. Esto produce comunmente la necesidad de "desaprender" para poder re-aprender correctamente los conceptos. Incluso B. Meyer menciona un método llamado "golpeelos dos veces" para atacar este fenómeno [Meyer97 pp. 935]).

El análisis y diseño orientado a objetos permiten modelar en un lenguaje gráfico la arquitectura de un sistema. Es decir, en los diversos diagramas que existen (de sistemas/subsistemas, clases, objetos, interacción, estados, etc.) se plasman todas las decisiones en cuanto a datos y comportamiento, relaciones entre clases e interacciones entre objetos. Esto permite examinar, discutir, comparar y reevaluar la arquitectura de un sistema de una manera mucho más accesible y sencilla que en código directamente, o con descripciones textuales del mismo. El análisis y diseño orientado a objetos no es el único método que logra esto, pero si presenta ventajas fundamentales frente a otros métodos de análisis y diseño al manejar los datos y el comportamiento juntos en las ya conocidas clases. Se puede decir por lo anterior, que el lenguaje (gráfico) por medio del cual se modelan los sistemas orientados a objetos, permite un nivel mayor de abstracción sobre la arquitectura de sistemas.

¿Cómo puede saber un programador inexperto si una decisión que está tomando sobre la arquitectura, el diseño o la implementación es buena? Normalmente cada programador resuelve el problema a su manera, según la poca o mucha experiencia que tenga. Su experiencia le permite utilizar una solución que sabe que funcionó en el pasado.

Un modelo orientado a objetos, sea cual fuere, si es examinado con detenimiento, casi invariablemente muestra repeticiones en las relaciones y/o colaboraciones entre sus objetos. Si son analizadas con detenimiento dichas repeticiones, en una gran medida pueden ser "factorizadas". Esta factorización por un lado reduce substancialmente la complejidad, y da por resultado *candidatos* posibles a patrones. ¿Por qué sólo candidatos? Porque para que un conjunto de clases u objetos y sus colaboraciones sean consideradas patrones, es necesario que se repitan en varios sistemas. Los patrones entonces surgen de arquitecturas o modelos existentes, de objetos ya plasmados en un diseño o existentes en un sistema funcionando. Son el resultado de análisis e introspección profunda, y de "reconocer" (gracias a la experiencia y el

conocimiento de varios sistemas) repeticiones de conjuntos de objetos y colaboraciones entre ellos[2].

Los patrones son el siguiente paso natural a la programación orientada a objetos. Permiten en forma concisa plasmar y transmitir "buenas" arquitecturas, diseños o decisiones de implementación. Deben ser el resultado de la experiencia de mucho tiempo de programación. De hecho, en la conferencia PLoP (Pattern Languages of Programs), solo son aceptados los patrones que hayan sido utilizados al menos en dos sistemas diferentes.

Algunos de los beneficios de los patrones son:

- Capturan decisiones de diseño que son recurrentes y que han mostrado ser buenas
- Permiten simplificar sistemas gracias a la factorización de elementos de la arquitectura
- Mejoran la comunicación
- Permiten expresar un cúmulo enorme de experiencia de una forma muy concisa y concreta
- Permiten almacenar y catalogar esta experiencia
- Permiten la comunicación entre programadores o arquitectos de sistemas de dicha experiencia, especialmente con los principiantes
- Proveen un vocabulario común de principios de diseño, que mejora la comunicación entre programadores o arquitectos de sistemas al permitir que se dé a un nivel mayor de abstracción
- Al ser independientes de la metodología de diseño y del lenguaje de programación permiten transferir la experiencia a otras áreas de la computación inclusive bajo paradigmas de programación diferentes
- Permiten documentar un sistema de manera concisa y completa
- Permiten la evolución más rápida de los diseños de sistemas nuevos ya que la experiencia colectiva y de otros programadores no presentes, interviene en las decisiones del nuevo sistema
- Permiten la reutilización de código, diseños y arquitecturas

Algunos problemas que persisten

- ¿Cómo buscar y cómo encontrar el patrón que necesitamos?
- Es necesario invertir mucho tiempo en su estudio (aunque eso finalmente puede ser utilizado en más de un proyecto), y muchas veces en el momento en el que los necesitamos no contamos con el tiempo suficiente para profundizar en su estudio
- No existe un catálogo general de patrones, por lo que no se puede saber si ya existe algún patrón escrito sobre un tema determinado o no. Esto nos obliga a recurrir de nuevo a expertos, o solucionarlo con nuestros recursos mientras que probablemente existe publicada una solución mejor y más atinada que la nuestra
- No existe un formato establecido para la escritura de patrones. Existen varios formatos cuyos respectivos seguidores defienden celosamente, por lo que será muy difícil llegar

[2] En esta dirección también han sido publicadas heurísticas de programación orientada a objetos. En estas heurísticas se presentan buenos(as) estilos, costumbres y criterios genéricos de diseño y programación orientada a objetos que ayudan a mejorar rápidamente los estilos de programación y cumplen funciones parecidas a los patrones, pero en un nivel más genérico (y por tal menos concreto) que los patrones. Se recomienda la lectura de [Riel96].

a unificarlos. Lo anterior impide almacenarlos y clasificarlos de manera uniforme, y poder realizar búsquedas por algún atributo determinado (por ej. contexto, usos conocidos, fuerzas, etc.).

Aunque los patrones aún tengan estas desventajas sin duda son un paso muy importante en la evolución de la programación orientada a objetos (aunque no sean exclusivos de ella, de ella surgieron) y son una herramienta didáctica muy valiosa para la adquisición rápida y efectiva de conocimientos que vienen de la experiencia de años de trabajo de expertos en el campo.

En este trabajo se presentan tres patrones que forman parte de un modelo para sistemas administrativos que, al ser presentados como patrones permiten, por un lado, ser utilizados independientemente uno del otro en aplicaciones de otros dominios y por otro lado, presentar el modelo de manera muy concisa.

El descubrimiento de los dos primeros patrones fue algo muy interesante. Inicialmente se estaba desarrollando un sistema para una industria textil, y ya concluido el análisis y diseño, se comenzó a analizar un segundo sistema (para control y administración de agentes de seguros). A pesar de lo alejados que son (aparentemente) estos dos dominios, se encontró que en el corazón de los sistemas todo era totalmente común. Existían personas que jugaban más de un rol diferente en el sistema, los documentos requerían relacionarse con otros documentos y con personas, y los puntos de mayor posibilidad de cambios en el sistema eran justamente en estas relaciones entre documentos y roles. De ahí surgió en ambos sistemas un diseño muy similar para atacar este problema.

El descubrimiento del tercer patrón fue en el primer sistema. Al analizar en la empresa textil (la del primer sistema) la necesidad de manejar en sus almacenes cantidades de tela provenientes de distintos puntos del planeta, y por lo tanto en diferentes unidades de medida, se diseñó una estructura de clases que posteriormente surgió idéntica (incluso la manera en que se dibujaron las clases participantes) al resolver las conversiones entre diferentes monedas de los importes de facturas, reportes, etc.

Estos tres patrones fueron presentados en la conferencia PLoP '96 [Schoenfeld96] (septiembre de 1996), y fueron muy bien recibidos. En particular el patrón de "Valores Convertibles", fue comentado con Martin Fowler, quien lo había descubierto también de manera totalmente independiente prácticamente al mismo tiempo, y de hecho estaba en proceso de publicación en su libro, que salió a la luz en enero de 1997, 4 meses después de la conferencia. El haber sido descubierto simultáneamente en lugares y países diferentes y sin tener ningún contacto, reafirma que es un patrón muy valioso, y de hecho está siendo utilizado en un gran número de sistemas, y seguramente lo seguirá siendo en muchos más por venir.

En septiembre de 1997, fue publicada una versión más completa y corregida del patrón "Personas y Roles" en ENC'97 [SO97].

El patrón de Personas y Roles de hecho fue re-publicado por el mismo Fowler en PLoP '97 bajo el título de "Dealing with Roles". En dicho artículo compara este patrón con otras soluciones que han sido utilizadas comúnmente en aplicaciones comerciales pero que presentan deficiencias que convierten a este patrón en una de las soluciones más ventajosas.

1. Patrones

Definición

¿Qué es un patrón? No existe hasta el momento un consenso respecto a la definición de patrón. La definición de Alexander en "The Timeless Way of Building" [Ale79] (pp. 247) es la siguiente:

"Cada patrón es una regla con tres partes, que expresa la relación entre un contexto, un problema y una solución.

Como un elemento en el mundo, cada patrón es una relación entre un determinado contexto, un determinado sistema de fuerzas encontradas que ocurren repetidamente en ese contexto, y una determinada configuración del espacio que permite la solución entre estas fuerzas. [...]

El patrón es por lo tanto, al mismo tiempo una cosa que sucede en el mundo y una regla que nos dice como y cuando crear esta cosa. Es tanto una cosa como un proceso; tanto una descripción de la cosa, como una descripción del proceso que crea dicha cosa."

Por otro lado, una de las últimas definiciones dadas es:

"Los lenguajes de patrones (Pattern Languages) son un conjunto de patrones que proveen todas las soluciones de diseño de un determinado campo" [BMRSS96, pp. 422].

Historia

Aunque patrones han existido siempre en computación, estos nunca se habían escrito per-se bajo un formato determinado y con el objetivo de ser publicados por sí solos. Todo programador aprende empíricamente ciertos patrones a lo largo de su carrera, los cuales generalmente no comparte más que con un reducido grupo de colegas. La idea de hacer un catálogo de patrones para difundir buenas costumbres de diseño, o soluciones bien probadas es relativamente reciente.

En los sesentas, el arquitecto Christopher Alexander y un equipo del "Center of Environmental Structure" en Berkeley, California trabajaron durante más de 20 años desarrollando un sistema de más de 250 patrones para la arquitectura. Estos patrones permiten resolver diversos problemas de diseño arquitectónico, desde pueblos hasta decoración de interiores. Ward Cunningham y Kent Beck, dos de los pioneros de Smalltalk, fueron los que al ser inspirados por el trabajo de Alexander, crearon cinco patrones sobre interfaces de usuario [Coplien95] que marcan el nacimiento de los patrones en la ingeniería de software.

La primera publicación sobre patrones fue el doctorado de Erich Gamma [Gam91] en 1991, y no fue ampliamente reconocida fuera de Europa central. Más de la mitad de los patrones del libro que dio mayor popularidad a los patrones de diseño del "Gang of Four (GOF)" [GHJV 95], estaban ya descritos en esta tesis de doctorado.

Entre otras, algunas de las figuras más sobresalientes del ramo son: James O. Coplien [Coplien94] [Coplien95] (idioms en C++, patrones de organización), Douglas C. Schmidt [PloP3 S] (patrones sobre sistemas distribuidos y concurrencia), Robert Martin [PLoP1 M] (C++), Peter Coad [Coad95] (patrones de análisis y diseño orientado a objetos), Wolfgang Pree [Pree94] (patrones en desarrollo de frameworks), Martin Fowler [Fow96] (patrones del dominio del problema (aplicaciones para hospitales e instituciones financieras), aunque su libro lleva como título "Patrones de Análisis") y Frank Buschmann [BMRSS96] (Patrones de arquitectura de sistemas).

Aunque hoy en día existen ya una gran cantidad de libros, listas de correo, e incluso un congreso completamente dedicado a patrones evidentemente es un área de la computación muy verde y que aún le espera un largo camino por recorrer.

Formato

Existen dos formatos principales bajo los cuales se escriben la gran mayoría de los patrones. El primero es el de C. Alexander que consta de tres partes principales aparte del nombre que es extremadamente importante, pues de él depende poder registrarlo en la memoria y asociarlo fácilmente a los conceptos referidos en el patrón. Las tres secciones son:

- Contexto – Donde se plantea el entorno existente para el cual se aplica el patrón
- Problema – Donde se describe el problema a resolver por el patrón
- Solución – Propuesta al problema bajo el contexto dado

El formato de Alexander de tres secciones posteriormente es extendido para incluir otras secciones más, como son:

- Fuerzas – Los elementos que están en juego y que la solución dada provoca que se obtengan o sacrifiquen (por ejemplo eficiencia, complejidad de la implementación, facilidad de extender o generalizar, flexibilidad, etc.).
- Contexto resultante – Situación resultante después de aplicar la solución
- Sustento Lógico (Rationale) – Razonamiento de lo que pasa en el fondo con el problema o solución. Es un marco teórico que intenta explicar por qué sí funciona la solución propuesta, o por qué tiene las ventajas o desventajas que se mencionan.

El segundo, el formato del GOF con 12 secciones principales:

- Intención – Incluye brevemente qué es lo que hace el patrón, y cuál es el problema a resolver
- Conocido como – Sinónimos o apodos del patrón (si existen)
- Motivación – Un escenario donde se muestra con un ejemplo cómo el patrón resuelve el problema en "Intención"
- Aplicabilidad – Describe las situaciones donde es o no conveniente aplicar el patrón
- Estructura – Una representación gráfica en alguna notación conocida de diseño orientado a objetos que describa las clases involucradas y sus relaciones
- Participantes – Descripción de las clases involucradas y sus responsabilidades
- Colaboraciones – Descripción de cómo los participantes colaboran
- Consecuencias – Lo que se gana y pierde de usar este patrón. Aquí se incluye un poco lo que son las fuerzas en el formato anterior.
- Implementación – Indicaciones, recomendaciones y avisos que se deben de considerar al implementar el patrón. También se mencionan aspectos particulares de algún lenguaje que facilita o dificulta la implementación.
- Código ejemplo – Ejemplo en código de una implementación del patrón
- Usos conocidos – Ejemplos de la vida real de usos de este patrón

- Patrones relacionados – Otros patrones que se pueden utilizar conjuntamente con este patrón, o si tienen parecido, cuáles son sus diferencias.

Aunque hoy en día se cuestiona si se debe seguir rigurosamente cualquiera de estos formatos, sin duda alguna, el apegarse a cualquiera de ellos facilita su escritura pues no permite dejar fuera al menos los aspectos más importantes (especialmente en el segundo formato), y su lectura ya que permite al lector saltar de sección en sección según sea su interés en particular.

El formato propuesto por Peter Coad [Coad95] no ha sido utilizado más que por él mismo, por lo que no se considera haber sido bien aceptado en la "comunidad de patrones".

Los patrones deben de ser autocontenidos, es decir, los patrones se presentan como si fueran a ser publicados en un catálogo de patrones, por lo que deben definir y explicar todo lo que se requiera para su comprensión.

Clasificación

Los patrones pueden ser clasificados desde diferentes puntos de vista o dimensiones de clasificación, dependiendo de la perspectiva con la que se miren. Una dimensión es según su propósito: existen dos tipos principales de patrones: los patrones generativos y los de referencia.

Los patrones generativos son sistemas de patrones en los cuales existen uno o varios patrones de inicio, los cuales al igual que todos los patrones del conjunto, tienen referencias a patrones subsecuentes. A partir del patrón de inicio que se elija, se tienen varios caminos posibles a seguir según los patrones que éste apunte. El lector puede escoger el siguiente patrón que desee, y de éste, según las referencias, el siguiente y así sucesivamente. Este tipo de patrones genera un espacio de diseños o arquitecturas de sistemas que solucionan los problemas planteados en la sección de "problema" del lenguaje de patrones. Los patrones de Alexander son principalmente de este tipo.

Por otro lado, los patrones de referencia, normalmente atacan un problema específico, y aunque tienen referencias a otros patrones, se utilizan normalmente para solucionar problemas específicos de diseño o implementación y no para generar toda una arquitectura o diseño completo de un sistema.

Otra dimensión para la clasificación de patrones es su granularidad. Aunque no existe aún un consenso acerca de como clasificar los patrones en esta dimensión, Booch los clasifica en tres niveles (frameworks, mechanisms e idioms) [Booch96] que de hecho es una clasificación similar y equivalente a la de Buschmann et. al. [BMRSS96]:

- Arquitectura de sistemas - Expresa la organización estructural fundamental del sistema. Provee un conjunto predefinido de subsistemas con sus respectivas responsabilidades y las reglas para organizar sus interrelaciones.
- Patrones de Diseño - Describe una estructura recurrente de componentes con comunicación entre ellos, que resuelve un problema general de diseño bajo un contexto dado.
- Idioms - Patrón de bajo nivel específico de un lenguaje de programación. Describe como atacar aspectos particulares de implementación de componentes o sus relaciones explotando los recursos de dicho lenguaje.

Existen muchos tipos de patrones, y cada vez hay más. Hay patrones del dominio del problema [Fow96] (para un tipo específico de sistemas, en este caso del ámbito médico y financiero)

patrones de organización [Coplien95] (plantean esquemas organizacionales para el desarrollo de software), patrones sobre como escribir patrones [PloP3 MD], y al paso del tiempo seguramente se irán atacando nuevas disciplinas o áreas del desarrollo de software.

2. El modelo orientado a objetos de empresas

El modelo para empresas se describe en base a tres patrones: "Personas y Roles", "Documentos y Roles de Personas", y "Valores Convertibles". Estos tres patrones describen los componentes básicos para la construcción de sistemas para empresas (y en realidad para cualquier sistema administrativo de casi cualquier institución o empresa). Es importante resaltar como con base en estos tres patrones, se pueden generar sistemas verdaderamente sofisticados y complejos, que cumplan con las necesidades de registro y consulta de información de dichas empresas.

Los primeros dos patrones definen las bases de la arquitectura para este tipo de sistemas. El tercer patrón presenta una solución al registro de valores que pueden ser representados en diferentes escalas, si esto es relevante para el sistema.

El patrón de "Personas y Roles" resuelve el problema de que una misma persona puede jugar simultáneamente varios roles en un sistema. Por ejemplo, una persona puede ser empleado, y simultáneamente proveedor, cliente, cobrador, vendedor, transportista o cualquier otro "actor" para una compañía. Un primer acercamiento para la creación de estos objetos, sería que todos heredaran de persona (pues a simple vista un cliente *es_una* persona, al igual que un proveedor, empleado, etc. Ver la Fig. 1 Implementación no recomendada de roles de una persona.). El problema de realizar la jerarquía de esta manera, es que una vez creado un objeto empleado, si se desea ver como vendedor o cobrador (suponiendo que es la misma persona) sería necesario realizar una conversión del objeto empleado en vendedor o cobrador. Esto es problemático ya que por un lado, no es soportado de manera natural por la mayoría de los lenguajes orientados a objetos y por otro lado, si existen referencias al objeto empleado, al realizar esta conversión podrían crearse referencias inválidas pues estas asumen que apuntan a un empleado y no a un vendedor. Este problema se resuelve completamente, permitiendo que un objeto persona tenga y además conozca todos los roles que juega en el sistema.

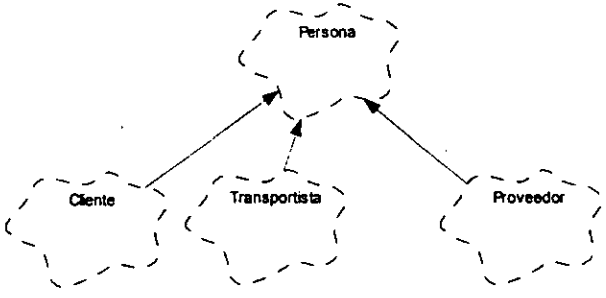


Fig. 1 Implementación no recomendada de roles de una persona.

El patrón de "Documentos y Roles de Personas", permite relacionar cualquier documento a cualquier número de roles de personas y documentos. Por otro lado permite relacionar a los roles de una persona a cualquier número de roles y documentos del sistema, permitiendo por medio de estos enlaces registrar todo lo relevante en la vida de un rol o de un documento.

Tres patrones de dominio del problema

Se presentan a continuación tres patrones, que pueden ser clasificados de la siguiente manera: respecto a su propósito: son de referencia (no son un lenguaje de patrones generativos). Respecto a su granularidad: son patrones de diseño, pues describen una estructura recurrente de componentes que resuelve un problema general de diseño de sistemas para empresas.

Respecto al formato que se decidió utilizar es el formato del GOF ya que incluye más secciones que los otros formatos, permitiendo exponer claramente todos los aspectos de los patrones por separado, facilitando la lectura del mismo.

2.1 Personas y Roles

Este patrón es la solución al problema descrito al inicio de esta sección, mismo que es encontrado en casi cualquier sistema de software orientado a objetos: un objeto puede jugar diferentes roles en un sistema, lo cual implica realizar cambios de tipo, algo no deseado pues la gran mayoría de los lenguajes de programación orientados a objetos no lo soporta. Utilizando una clase "Rol" como envoltura de la clase que requiere cambios de tipo, es posible solucionar este problema ya que dinámicamente es posible crear y destruir "Roles", sin afectar el tipo de la clase de la cual éstos son sus roles. El patrón se presenta a continuación:

Intención

Las personas en un sistema participan jugando diversos roles y pueden cambiar estos roles libremente sin que esto implique un cambio de tipo. Adicionalmente, una persona puede tener muchos roles simultáneamente incrementándolos o decrementándolos dinámicamente según sea la necesidad.

Motivación

Casi cualquier sistema de cómputo administrativo tiene relación con personas (usuarios, empleados, clientes, estudiantes, compañías, etc.). Estas personas intervienen en diversos procesos del sistema, en cada uno requiriendo atributos o comportamiento particular para ese proceso. Por otro lado, una misma persona puede incluso intervenir en varios procesos al mismo tiempo, por lo que requeriría obtener estos atributos o comportamiento dependiendo de los procesos en los que estuviera involucrada. Esto implica un cambio de tipo del objeto "persona" para poder jugar diferentes roles. Por Ej. un estudiante (que *es_una* Persona) puede en un momento dejar de ser estudiante para pasar a ser profesor (también *es_una* Persona) de un curso, y también ser el encargado del departamento de cómputo y tomar actividades administrativas para poder solventar sus estudios. Esto implicaría que una instancia de *Persona*, en un momento dado cambie a ser instancia de *Estudiante* y posteriormente pase a ser *Ayudante*, *Profesor*, etc. Es la misma persona pero debe aparecer en el sistema como instancias diferentes. El problema subyacente es que la herencia es una manera de clasificar que implica particiones: una instancia de un tipo, no puede ser simultáneamente de otro tipo de la jerarquía. Aún cuando se pudiera cambiar fácilmente de un tipo a otro de la jerarquía, el problema no estaría resuelto, ya que se puede ser *Estudiante* y *Ayudante* al mismo tiempo.

Lo que vemos es que las personas juegan diferentes Roles, incluso al mismo tiempo por lo que es necesaria otra clase (Rol) que nos permita modelar esto. Estudiante, Ayudante, Profesor, etc. pueden ser descendientes de Rol, lo cual permite que se creen y desaparezcan dinámicamente e incluso, que existan al mismo tiempo.

Aplicabilidad

Para definir los procesos de la empresa u organización es necesario primero identificar a los "actores" (individuos u organizaciones externas) que interactuan con la empresa, donde *interactuar* se entiende como recibir un servicio o dar un servicio a esta. Los procesos de la empresa son únicamente los que se originan o terminan con un "actor" [Jaco94]. Aparte de los actores (externos), dentro de la empresa cada departamento puede ser visto como cliente o proveedor de los otros departamentos. Esto le da un enfoque a la empresa que permite ver actores "internos" y se puede definir que los procesos de un departamento son únicamente los

que originan o terminan en un "actor interno". Todos estos actores tanto internos como externos son de hecho personas físicas o morales.

Por lo anterior, todo sistema de cómputo para empresas (y en este caso también organizaciones, instituciones, etc.) requiere representar a estos actores (personas físicas o morales) para poder realizar sus propias tareas dentro de los procesos administrativos de la misma.

Estructura

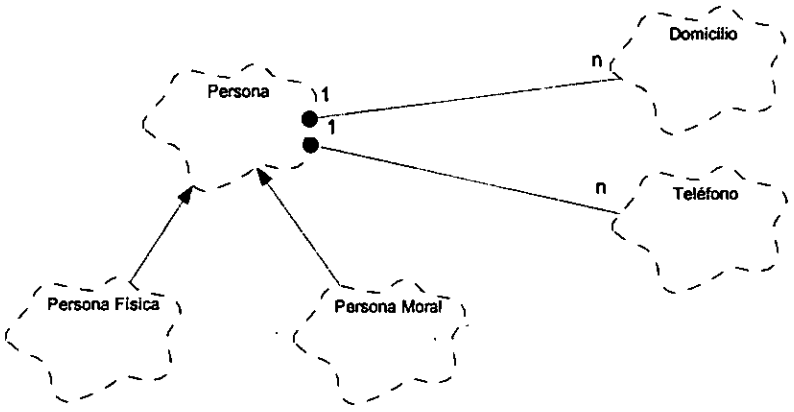


Fig. 2 Diagrama de Persona

Una persona puede ser física o moral.

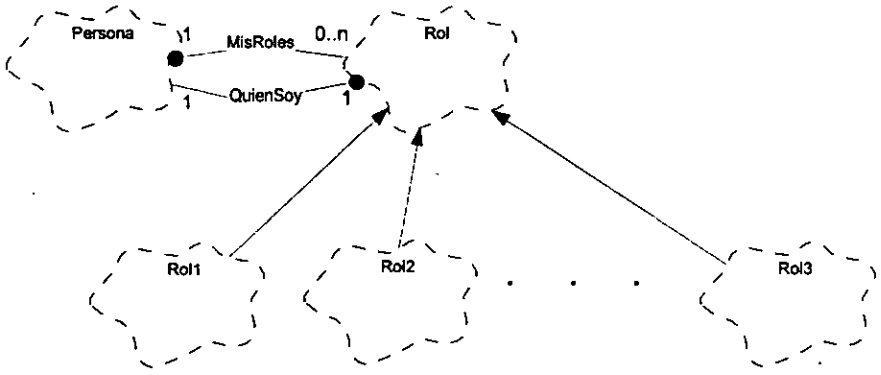


Fig. 3 Diagrama de Roles de Persona

Una persona puede tener 0 o más roles, y un rol tiene siempre una persona. Todo Rol hereda de *Rol*

Participantes

- *Persona* es la entidad que representa en el sistema a una persona Física o Moral. Tiene todos los atributos que no cambian de rol en rol y que son intrínsecos de la persona.
- *Rol* es la clase abstracta de la cual todo rol posible del sistema debe heredar. Cada *Rol* es responsable de tener los atributos y métodos que le sean necesarios para cumplir con su tarea específica dentro del sistema.

Colaboraciones

Una *Persona* tiene una lista de todos sus roles en el sistema. Por otro lado, cada *Rol* sabe de qué *Persona* es.

Rol asimila[3] [Kath97] a *Persona*, es decir, todos los atributos y métodos de una persona (nombre, RFC, teléfonos, domicilio, fecha de nacimiento, etc.) son accesibles desde todos los roles y pueden ser vistos como atributos propios de estos roles. Por ejemplo, un *Rol Empleado*, puede ver la fecha de nacimiento de la *Persona* para cálculos de nómina.

Los clientes del patrón (clases que utilicen el patrón) deben de accesar a la *Persona* siempre por medio del rol correspondiente. La *Persona* no debe de ser accesada directamente por otras clases para evitar dependencias rígidas entre las clases y siempre mantener los accesos por medio de estos intermediarios (envolturas) que son los Roles.

Consecuencias

Ese patrón evita la necesidad de cambiar el tipo de una instancia (*Persona*) cada vez que hay un cambio en el rol que juega dicha instancia.

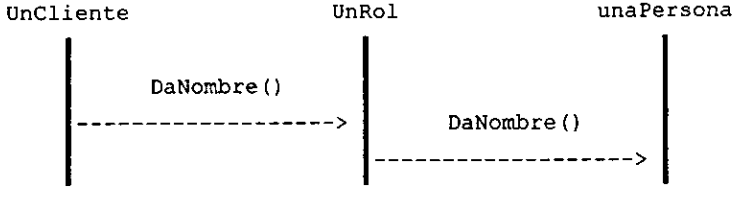
Ya que *Rol* asimila a *Persona*, estos pueden ser vistos como personas por su interfaz, mas no por su tipo, pues no heredan de persona.

Un objeto *Persona* puede tener inclusive dos roles del mismo tipo, por ejemplo una misma persona puede ser empleado en dos o más compañías simultáneamente (algo no tan extraño hoy en día) simplemente creando dos instancias de *Rol*, cada una con sus propios atributos como sueldo, fecha de contratación, etc.

Implementación

Ya que hoy en día los lenguajes de programación no soportan la asimilación, la implementación posible es que el *Rol* funcione como una envoltura de *Persona*, por lo que al ser solicitado un atributo, si lo tiene responde directamente, y si no, éste delega la petición a la persona y ésta da la respuesta correspondiente.

Por ej:



[3] La asimilación es un mecanismo de modelado de relaciones entre clases que mezcla características de herencia y contenido por referencia. Una clase A que asimila a la clase B, incorpora toda la interfaz de B, pero no hereda de ella, pues no pasa a ser de su tipo; simplemente incorpora a su interfaz, la interfaz de B. Si la interfaz de B sufre un cambio, este es automáticamente reflejado por A.

Es necesario realizar algunas validaciones para evitar situaciones contradictorias, como por ejemplo que una persona tenga el *Rol* de *Empleado* y *Desempleado* al mismo tiempo. Estas validaciones se deben hacer en los constructores de los roles, para evitar la creación de roles contradictorios.

Otra validación importante, es el que un rol puede existir solo para personas físicas o sólo para personas morales. Igualmente, la validación se debe de hacer en el constructor del *Rol* que tenga dicha restricción.

Es importante que en el sistema no se tengan referencias directas a la *Persona*, sino únicamente a los roles de esta, ya que la persona siempre esta jugando algún *Rol* específico en el sistema según el proceso donde intervenga. De esta manera, se asegura que para que una *Persona* participe en un escenario, debe tener dicho *Rol*, y si no lo tiene, simplemente no puede participar hasta que no se cree.

Uno de los parámetros del constructor de *Rol* es la *Persona* para la cual existe. Dentro del constructor esta referencia es hecha y en ningún otro lugar puede ser cambiada (un *Rol* no puede ser construido para una *Persona* y en un momento dado cambiar de *Persona*. No tiene sentido semánticamente hablando).

Usos conocidos

En el siguiente diagrama podemos ver algunos ejemplos de roles:

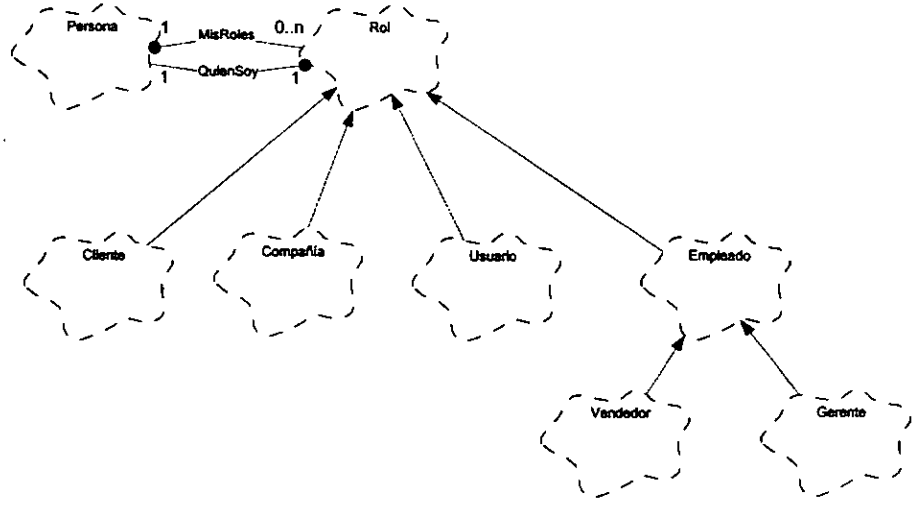


Fig. 4 Ejemplo del patrón de Personas y Roles

Este patrón es utilizado ampliamente en dos sistemas: uno para una compañía de textiles (KADIMA) y otro para un sistema para administración de agentes de seguros (DANIELS). En este último caso, fue de especial elegancia la solución, pues hay muchos roles que intervienen para un seguro. En un seguro participan el Beneficiario, el Asegurado, el Contratante (el que paga), etc. En algunos casos, todos son roles de la misma persona, y en otros, todas son personas diferentes. Es común que una persona que es beneficiario en una póliza de seguro, también es asegurado en otra. Ya que todos los datos de la persona fueron dados de alta para el

primer rol, es muy sencillo y rápido darla de alta para otro rol. Adicionalmente al saber todos los roles que tiene una persona, se pueden realizar cruces de información muy valiosos para la organización que los maneja.

Patrones relacionados

Martin Fowler llama "Party" ([Fow96] pp. 18) a la entidad que abarca personas tanto físicas como morales, la que en este patrón es "Persona" pero no incluye la cuestión de los Roles.

El patrón "Carta-Sobre" (Envelope-Letter) de Coplien [Coplien94] es una técnica para cambiar de clase a un objeto en tiempo de ejecución. Eso no es exactamente lo que se busca, ya que es necesario también tener varios roles de una persona **al mismo tiempo**, e incluso del mismo tipo.

El patrón de Estado (State) del GOF es un caso similar al anterior.

Nótese que este patrón podría ser generalizado para que una entidad cualquiera tenga varios roles. Por ejemplo, una Factura para Cobranzas es el documento para cobrar, mientras que para Inventarios es el documento para permitir la salida de mercancías. Estos son dos roles del mismo documento. El método OOram de hecho ataca este tipo de generalización pero con una perspectiva diferente [RWL95].

El método "Perspective Application" puede ser utilizado en el análisis para la identificación de roles de personas [Liao96]

James Odell y Conrad Bock analizan los roles como un tipo de asociación más (relación) entre objetos. Desde este punto de vista, los roles son tanto los objetos (tal y como se presentan en este patrón) como la relación implícita por ellos entre la persona y otros objetos donde este rol participa. En su artículo describen como esta relación descrita en el análisis con solo una línea entre dos objetos, de hecho en la implementación implica múltiples mapeos ("mappings") entre objetos, dependiendo de la complejidad de la relación [BO98].

2.2 Documentos y Roles de Personas

Las empresas hoy más que nunca, constantemente sufren cambios en su manera de trabajar y en su organización, debido a una necesidad de adaptación a un entorno muy dinámico. De contar con un modelo orientado a objetos que representara un mapeo de la realidad, los cambios que sufriese la empresa podrían ser reflejados fácilmente en dicho sistema. Un "buen" mapeo de la realidad implica que cambios pequeños de ésta producen cambios pequeños en el modelo. Si esto se cumple, podemos asegurar la facilidad de modificación del sistema y su resistencia a los cambios.

El patrón que se presenta a continuación, da un fundamento para el modelaje de cualquier organización donde intervienen personas y documentos.

Intención

Expresar cualquier relación entre *Documentos* y *Roles* de *Personas* en un sistema administrativo. Dichas relaciones pueden ser modificadas y validadas fácilmente para permitir que el sistema se adapte rápidamente a cambios en estas relaciones.

Motivación

En una organización, la mayoría de las operaciones realizadas deben ser registradas, administradas, archivadas, verificadas, etc. y para ello se utilizan documentos. Estos documentos siempre se relacionan con las personas que los producen, utilizan, revisan o

reciben. A su vez, cada documento es el origen o consecuencia de otro(s) documento(s) a lo largo de los procesos internos de una organización.

Toda empresa o institución constantemente sufre cambios en su organización interna y en relación a los actores externos con los que interactúa, generando la necesidad de modificar su sistema de cómputo para adaptarse a dichos cambios. Generalmente estos cambios son difíciles de hacer y se requiere mucho tiempo para llevarlos a cabo, lo cual repercute en el tiempo de respuesta a los cambios de dicha empresa, y por lo mismo en su adaptabilidad y competitividad.

Los siguientes son solo algunos de los factores que provocan los cambios en las empresas:

1) Se pasa por un proceso de reingeniería de la organización
2) Se descubren fallas en la organización y se corrigen
3) Incorporación de nuevas tecnologías (código de barras, tarjetas magnéticas, sensores, etc.).
4) Cambios en filosofías administrativas que reducen o aumentan la supervisión
5) Crecimiento o contracción de la organización.
6) Regulaciones o desregulaciones a las que están sujetas las empresas
7) Cambios en el mercado
8) Cambios por competencia (innovaciones, necesidad de igualar servicios, etc.)

Los cambios provocados en la organización por los puntos anteriores, afectan: a) La existencia misma de roles de personas y documentos, b) Las relaciones que tiene un documento con otros documentos, c) Las relaciones que tiene un documento con roles de personas (en quienes y en qué orden intervienen) y d) Las relaciones que tienen los roles de personas entre sí.

Este patrón da una respuesta a la necesidad de realizar dichos cambios fácil y rápidamente.

Aplicabilidad

Se puede utilizar este patrón para el desarrollo del sistema de cómputo de una empresa u organización donde existen personas que utilizan documentos como apoyo o base para la realización de sus actividades y se dé al menos alguna de las siguientes situaciones:

- Cuando se requiera de flexibilidad y adaptabilidad al cambio en los procesos entre los actores tanto internos como externos.
- Cuando la organización está sujeta a alguno de los factores que aparecen en motivaciones
- Cuando la organización está naciendo y aún no se cuenta con una estructura consolidada

Estructura

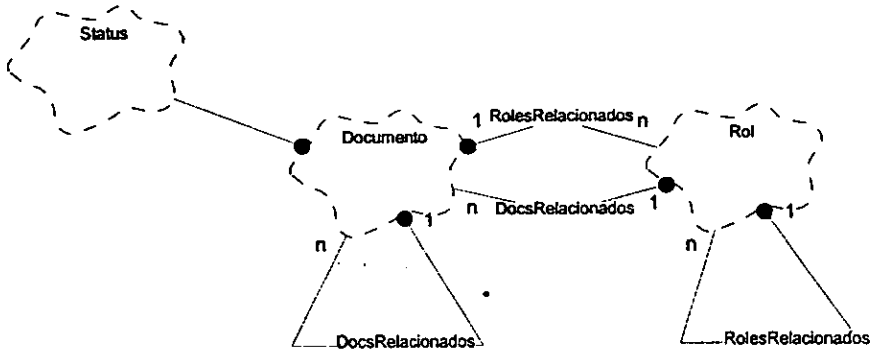


Fig. 5 Patrón de Documentos y Roles de Personas

Participantes

- *Documento*. Son los objetos más comunes en los sistemas administrativos. Todo documento (*Carta, Póliza, Factura, Cheque, Reporte, Memo*, etc.) debe heredar de *Documento* y mapea a un documento real de la empresa o institución. El *Documento* es el responsable de validar sus cambios de status y permitirlos o no. En estos cambios de status es donde se validan las relaciones requeridas con otros documentos o roles.
- *Rol*. Es un rol de persona, como se describe en el patrón "Roles de Personas".

Colaboraciones

Las colaboraciones son muy sencillas desde el punto de vista de la programación. Existen 4 métodos en cada clase (*Rol* y *Documento*) que permiten relacionarlas entre sí:

`LigaConDocumento(Documento)`

`LigaConRol(Rol)`

`DesligaDeDocumento(Documento),`

`DesLigaDeRol(Rol)`

En base a estos cuatro métodos se realizan todas las ligas posibles entre documentos y personas. Según los procesos de la empresa o institución, estas interrelaciones son permitidas o no, lo cual define las colaboraciones finales entre estas clases, o más bien, entre los descendientes de estas clases.

Consecuencias

Con esta estructura es posible navegar por los documentos que están relacionados y por las personas involucradas. Por ejemplo, si llega a haber un problema con un Cliente dado, es posible consultar todos los documentos relacionados con él. Digamos que encontramos el documento con el problema, de este documento es posible saber quién inició ese documento, qué otros documentos (como cartas) fueron enviados o utilizados, a quién se le enviaron, quién realizó la comunicación, quién visitó al cliente anteriormente y por lo tanto, con quién es necesario hablar y qué otros documentos también están involucrados.

Este patrón da respuesta a los factores que provocan los cambios en los sistemas de las empresas mencionados en la sección de motivación, ya que permite la fácil creación/eliminación de documentos y roles tanto de personas (todo documento de la empresa

hereda de *Documento*, por lo que no es necesario redefinir toda la complejidad de sus interrelaciones, al igual que con los herederos de *Rol*) como de documentos, dándole una enorme flexibilidad y adaptabilidad al sistema.

Implementación

Es importante la validación de las relaciones permitidas entre roles, entre documentos, y entre roles y documentos. De igual manera, es importante tener mecanismos que requieran o exijan dichas relaciones en ciertos momentos determinados.

Respecto al documento, el control se da por medio de los cambios de status del mismo. El documento tiene un atributo Status y tres métodos:

- CambiaStatus(NuevoStatus) - público, llamado por el que necesite cambiar el status del documento.
- ValidaCambioDeStatus(NuevoStatus) - protegido y virtual, llamado por CambiaStatus(NuevoStatus) para validar si es posible realizar ese cambio de status.

En la validación del cambio de status, se realizan todos los chequeos necesarios de *Roles* o *Documentos* requeridos, para que estos ya se encuentren ligados al mismo.

Existe una semántica intrínseca en estas relaciones. Por ejemplo, un Rol Cliente relacionado a un Documento Factura, implica que es el cliente para el cual esta hecha la factura. Un cliente relacionado a un Cheque, nos dice quién hizo el cheque, mientras que un Proveedor relacionado con un cheque nos dice para quién es ese cheque (obviamente es necesario incluir casos en los que la empresa hace un cheque a un cliente por una devolución, etc. pero estos casos pueden ser identificados ya que la empresa reconoce cuales son sus cheques pues ella misma los expide, y conoce la cuenta de la que salieron). Esto simplifica la estructura pues no es necesario describir en otro lugar que tipo de relación se tiene, ésta está descrita por el tipo de los Roles.

Los cuatro métodos mencionados en colaboraciones, realizan la liga o separación entre un *Documento* y un *Rol* dado, o entre dos *Documentos* y *Roles* dados, realizando todas las actualizaciones en ambos lados (si se liga un *Documento* a un *Rol*, esta liga debe aparecer tanto del lado del *Rol* como del *Documento*).

Si la implementación de la persistencia se realiza por mapeo a una base de datos relacional, el desempeño impide la implementación del patrón tal y como es expuesto en el presente documento por la necesidad de construir múltiples uniones (joins) lo cual repercute fuertemente en el tiempo de respuesta del sistema. Es necesario contar con una base de datos orientada a objetos para que no exista penalización alguna al tener tantas ligas en una estructura con tantas relaciones cruzadas.

Usos conocidos

Un ejemplo puede ser el siguiente: una carta *es_un Documento* que tiene un *Remitente* y un *Destinatario* (ambos roles de personas). Cuando la carta se crea, está en el status *Inicio*. Cuando se graba por primera vez, cambia su status a *Edición* y para este cambio, ya exige tener relacionado a un *Remitente* y a un *Destinatario*. Posteriormente cuando se envía cambia su status a *Enviada*, lo cual evita que pueda seguir siendo modificada. Al ser asignados los roles *Destinatario* y *Remitente*, en ellos mismos se actualiza su lista de *DocsRelacionados*, por lo que les aparece ahora la carta como un documento que enviaron (caso *Remitente*) o recibieron (caso *Destinatario*). Si dicha carta es una *Cotización*, y posteriormente se realiza la venta, entonces se relaciona con la *Factura*, la cual se registra para tener un antecedente de dicha venta.

Este patrón fue utilizado en dos sistemas: uno para una compañía de textiles (KADIMA) y otro para un sistema para administración de agentes de seguros (DANIELS). En ambos sistemas no

se cuenta con una base de datos orientada a objetos, por cuestiones de desempeño fue simplificado fuertemente, y no se cuenta actualmente con una versión completa del patrón en dichos sistemas. Estos sistemas implementaron la persistencia de objetos sobre una base de datos relacional, lo cual impide en la práctica realizar una cantidad grande de Joins, impidiendo la aplicación completa del patrón.

Patrones relacionados

El patrón "Personas y Roles" está íntimamente relacionado con éste patrón.

2.3 Valores Convertibles

Este patrón es muy útil cuando es necesario realizar conversiones entre monedas, medidas, etc. El patrón describe qué clases son necesarias para poder salvar los datos como fueron originalmente generados preservando la precisión, y cómo pueden ser accesados fácilmente en cualquier otra escala, sin peligro de cometer errores de conversión ya que las mecánicas de conversión están encapsuladas (ocultas) en objetos especializados.

Intención

Manejar y guardar cantidades en su formato original (para tener buena precisión), pero permitiendo ser utilizadas transparente y fácilmente en cualquier otra escala o representación.

Motivación

Es común tener que manejar cantidades que originalmente son registradas en una unidad de medida o escala, pero posteriormente deben ser expresadas en otra escala o unidad. Por ejemplo, si se guardan cantidades o medidas, estas pudieron ser tomadas inicialmente en una unidad, pero posteriormente se requiere que sean expresadas en otra unidad. Una solución es realizar desde un principio, la conversión de todas las medidas para que queden en la unidad deseada. Los problemas de esta solución, son en primer lugar, el cómo definir en qué unidad guardar todos los datos. En este momento, puede desearse guardar en una unidad que posteriormente se quiera cambiar. Es una decisión que nos ata desde un principio a algo que es muy probable tenga que cambiar en el futuro. De cualquier manera, si en el futuro se quisiera cambiar de unidad de medida, se podrían realizar las conversiones pertinentes, pero esto a su vez, provoca que haya pérdida en la precisión por tantas conversiones con valores de punto flotante y las limitantes conocidas de las computadoras en este sentido.

Aplicabilidad

Utiliza el "Patrón de Valores Convertibles" cuando sea necesario expresar datos en escalas diferentes a la original con la que fueron registrados o, si es necesario comparar relacionar o realizar operaciones aritméticas sobre datos que pueden estar en unidades o escalas distintas entre sí.

Esto obviamente, sólo es posible si los datos pueden ser expresados en diferentes unidades o "sistemas de medida" y pueden ser traducidos de una a otra.

Por ejemplo, una compañía que tiene sucursales en diferentes países, puede necesitar ver sus resultados financieros expresados en dólares para compararlos, aún si estos fueron originalmente almacenados en pesos, yens, libras, etc.

De la misma manera, la misma compañía puede comprar y vender productos procedentes de distintos países, donde son manejados en unidades de medida diferentes. Al realizar una compra de un país y venta en otro, es necesario realizar la conversión para de esa manera

entregar exactamente lo que se requiere según el país comprador. Además, esta compañía puede consolidar sus inventarios en una sola unidad de medida para costearlos, y conocer así su valor real respecto a una unidad de medida y una moneda (metros y dólares, o pies y libras, etc.).

También puede ser aplicable a sistemas donde se registran mediciones tales como sistemas médicos, de monitoreo, para apoyo en experimentos científicos, etc.

Estructura

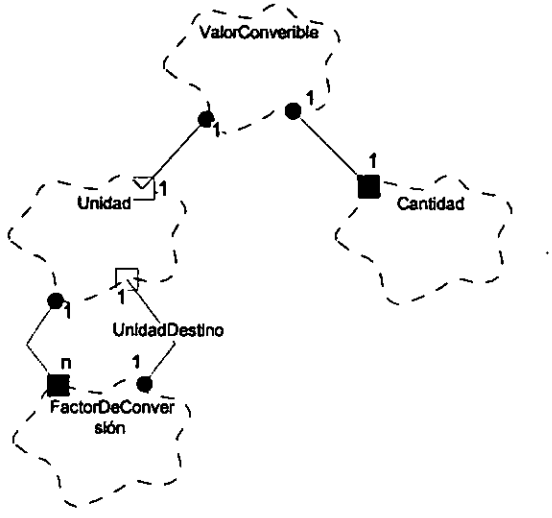


Fig. 6 Patrón de Valores Convertibles

Cada *ValorConvertible* tiene una *Cantidad* y una *Unidad.*

Cada *Unidad* tiene "n" *FactoresDeConversión* a otras unidades, para poder realizar las conversiones.

Un *FactorDeConversión* tiene una referencia a la unidad destino a la cual se convierte gracias al factor que contiene.

Participantes

- *ValorConvertible* - Es la clase que contiene el valor que puede ser convertido, es decir, representado en otra escala. Ésta es la clase que se utiliza realmente en el sistema dentro de los documentos, registros, productos, etc. que tienen un valor a registrar. A esta clase es a la que se le pide el valor en la unidad que se desee.
- *Cantidad* - Guarda el valor numérico, generalmente un decimal o punto flotante.
- *Unidad* - Guarda el tipo o unidad original de este valor convertible. Esta es la clase responsable de realizar las conversiones necesarias entre diferentes unidades o escalas. Para realizar las conversiones, utiliza a la clase *FactorDeConversión.*
- *FactorDeConversión* - Es la clase que guarda la información que permite realizar las conversiones. Una *Unidad* puede tener todos los factores de conversión para todas las otras unidades o sólo para algunos de interés.

Colaboraciones

A un objeto *ValorConvertible* se le puede solicitar su valor en una unidad cualquiera que puede ser o no la original con la que se almacenó. En la siguiente figura se presenta el diagrama de interacción para realizar la conversión necesaria:

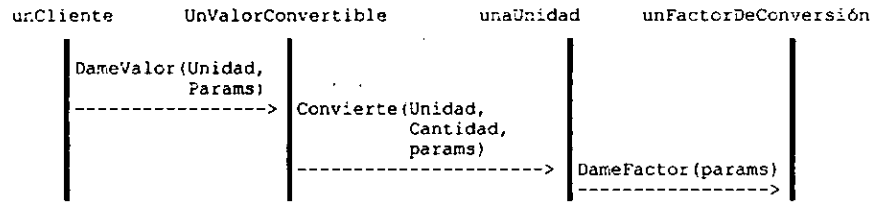


Fig. 7 Ejemplo de interacción para dar un valor en otra unidad a la almacenada

Si la unidad solicitada es la misma que la del *ValorConvertible*, directamente responde con lo que tenga en *Cantidad*. De no ser así, manda una petición de conversión a su unidad, la cual a su vez, solicita al factor correspondiente el factor de conversión hacia la unidad destino.

Los parámetros adicionales que se presentan son para el caso en que se requieren otros datos para realizar la conversión, como por ejemplo en conversiones entre diferentes monedas, donde es indispensable pedir la fecha (y hora en algunos casos) para realizar la conversión, pues los tipos de cambio son variables respecto al tiempo.

ValorConvertible es la responsable de expresar su valor en cualquier unidad solicitada. Para esto se apoya en *Unidad*, la cual es responsable de realizar conversiones entre esa *Unidad* y otras permitidas. Para esto utiliza a la clase *FactorDeConversión* donde se guarda toda la información necesaria para realizar dichas conversiones.

Consecuencias

Los datos pueden ser almacenados tal y como fueron generados, y así preservar al máximo la precisión. Ya que las conversiones son realizadas dentro de *ValorConvertible*, los datos pueden ser accesados como si estuvieran en la unidad deseada y como si todos los *ValoresConvertibles* estuvieran en esa misma unidad, permitiendo manipular los datos de una manera totalmente uniforme y transparente. Por lo anterior, el mismo reporte (sea cual fuere) puede pedirse en cualquier unidad o escala deseada, sin que esto haya sido siquiera contemplado por el programador que hizo el reporte. Por ejemplo, cualquier reporte del sistema que trate sobre importes monetarios, puede ser solicitado en cualquier moneda sin que esto implique ninguna modificación especial del código. Incluso en monedas que no existían en el sistema cuando se creó el reporte. Lo único que se necesita es dar de alta la moneda, y sus tipos de cambio respecto a las monedas que ya existen en el sistema. Esto es especialmente útil, cuando surge una nueva moneda (Nuevos Pesos, UDIS, etc.) y cualquier reporte, documento, o consulta requiere ser expresado en ella.

Implementación

Cantidad puede ser cualquier tipo numérico, aunque es importante analizar la precisión necesaria según la escala y las conversiones que se fueran a realizar.

Como fue mencionado anteriormente, los parámetros de *DameValor*, *Convierte* y *DameFactor* pueden ser diferentes dependiendo del tipo de conversiones que se requieran.

Es posible que no todos los *FactoresDeConversión* sean capturados en el sistema, lo cual haría imposible calcular algunas de las conversiones. En estos casos es necesario definir de que

manera se manejará la situación (levantar una excepción, dar un valor default, ignorar el error y continuar, etc.). El patrón "Meaningless Behavior" del lenguaje de Patrones CHECKS [PloPl C] es muy apropiado para esto.

Pueden también existir conversiones ilegales, tales como centímetros a grados, o libras a metros. De igual manera deben definirse las acciones a realizar en estos casos, aunque normalmente lo correcto es levantar una excepción, ya que en este caso se está realizando una operación sin sentido. *Unidad* es la responsable de manejar estos errores.

La mayoría de las conversiones son simétricas, y esto debe de ser considerado, es decir, un valor convertido a otra unidad y luego de regreso a la unidad original debe ser el mismo.

Usos conocidos

Los dos usos más comunes de este patrón son en cantidades de inventarios (Fig. 8 Instancia del patrón de Valores Convertibles para cantidades de inventarios) y en importes monetarios. (Fig. 9 Instancia del patrón de Valores Convertibles para importes monetarios).

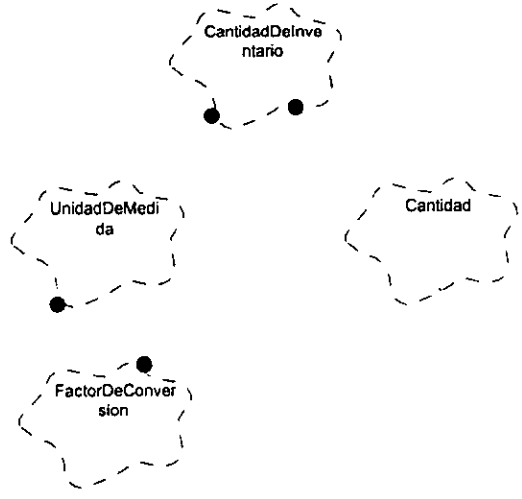


Fig. 8 Instancia del patrón de Valores Convertibles para cantidades de inventarios

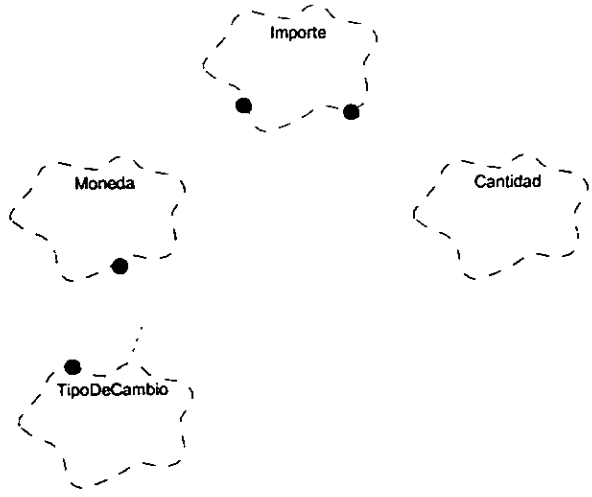


Fig. 9 Instancia del patrón de Valores Convertibles para importes monetarios

Patrones relacionados

"Units" de Martin Fowler [Fow96] es de hecho el mismo patrón.

Conclusiones

Sin duda alguna, los patrones presentan una excelente manera de registrar experiencia y conocimiento que permiten la futura explotación del mismo. Los beneficios que se tienen por mejorar la comunicación de dicha experiencia son incalculables y por lo mismo, permitirán acelerar la evolución de la ingeniería de software y mejorar la calidad del software. Algo indispensable hoy en día.

El formato que tendrán los patrones en el futuro no está definido. Los dos principales formatos tienen sus ventajas y desventajas. El formato del GOF no tiene las secciones de contexto y fuerzas, dos secciones muy valiosas en el formato de C. Alexander. Por otro lado, el formato de C. Alexander es muy genérico, con lo que fácilmente queden puntos flojos y ambigüedades en la descripción. Para una discusión mayor de estos puntos, se recomienda el artículo de [PloP3 MD].

Aunque el objetivo era encontrar un modelo para empresas, todos los patrones son aplicables a organizaciones, instituciones, empresas, etc., es decir, se tiene un espectro de aplicación mayor al esperado.

Los patrones no están en su versión final. De hecho, probablemente nunca lo estarán. En cada aplicación nueva de estos patrones, pueden encontrarse variantes que mejoren sus cualidades, o que extiendan sus alcances. Es un hecho que aunque son un punto de partida y ya están siendo utilizados en al menos dos sistemas, pueden ir siendo extendidos por otras personas en base a su propia experiencia e ideas. En ese sentido, no hay dueños de los patrones, sino toda la comunidad científica que los utilice y redescubra.

Cabe mencionar que el patrón de "Documentos y Roles de Personas" es muy poderoso, pero efectivamente presenta una desventaja de rendimiento si su implementación es sobre un mapeo a base de datos relacional. Esto no es un factor totalmente negativo, pues las bases de datos orientadas a objetos son una realidad hoy en día, y en dichas bases de datos, no existe problema alguno.

Referencias

[Ale79] C. Alexander. *The Timeless Way of Building*, Oxford University Press, 1979

[BO98] Conrad Bock, James Odell, *A more complete model of relations and their Implementation: Roles*, JOOP, Vol. 11 No. 2 (mayo. 98), pp 51-54.

[Booch94] Grady Booch. *Object Oriented Analysis and Design with Applications*, 2nd. ed., The Benjamin/Cummings Publishing Inc., 1994.

[Booch96] Grady Booch, *Managing the Object-Oriented Project*, Object Solutions Addison-Wesley, 1996.

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal, *A system of patterns*, John Wiley & Sons, 1996.

[CACM97-40,4] *The Debugging Scandal - And what to do about it*, Communications of the ACM, April 1997, Volume 40, No. 4, pp.27-29.

[Coad95] Peter Coad, *Object Models: Strategies, Patterns & Applications*, Prentice Hall, 1995.

[Coplien94] James O. Coplien. *Generative Pattern Languages: An emerging direction of software design*, C++ Report, Jul-Aug 1994.

[Coplien95] James O. Coplien. *The history of patterns*, http://c2.com/cgi/wiki?HistoryOfPatterns

[Fow96] Martin Fowler. *Analysis Patterns - Reusable Object Models*, Addison-Wesley, 1997

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

[Jaco94] Ivar Jacobson. *The Object Advantage. Business Process reengineering with Object Technology*, Addison-Wesley Publishing Company, 1994.

[Kath97] Ravi Kathuria. *Improved modeling and design using assimilation and property modeling*, JOOP, Vol. 9 No. 8 (ene. 97), pp 15-24.

[Liao96] S.Y. Liao, *Perspective Application: An Efficient Tool to Identify Roles and Subclasses in Object-Oriented Modeling*, ROAD Vol. 2, No. 5, Enero-Febrero 1996

[Meyer97] *Object Oriented Software Construction*, Prentice Hall International (Series in Computer Science), 1988 ¡X!

[Nygaard81] Kristen Nygaard, Ole-Johan Dahl: The Development of the SIMULA languages, in History of Programming Languages, ed. Richard L. Wexelblat, Academic Press, New York, 1981, pp. 439-493.

[PloP1 C] J.O. Coplien, D. Schmidt. The CHECKS pattern Language of Information Integrity in *Pattern Languages of Program Design*, Addison Wesley, 1995, pp. 145-155

[PloP1 M] Robert Martin, *Discovering Patterns in Existing Applications in Pattern Languages of Programming 1*, Addison Wesley, 1995, pp. 365-393

[PloP3 MD] Gerard Mezaros, Jim Doble, *A pattern Language for Pattern Writing in Pattern Languages of Programming 3*, Addison Wesley, 1998, pp. 529-574

[PloP3 S] Douglas C. Schmidt, *Acceptor and Connector in Pattern Languages of Programming 3*, Addison Wesley, 1998, pp. 191-229

[Pree94] Wolfgang Pree, Design Patterns for Object Oriented Software Development, Addison Wesley/ACM Press, 1994/1995

[RWL95] Reenskaug, T., P. Wold y O.A. Lehne *Working with Objects: The OORam Software Engineering Method*, Prentice Hall, 1995

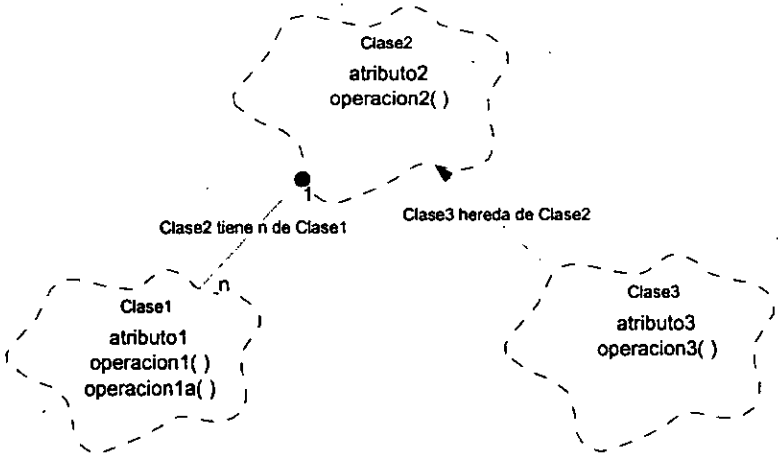
[Riel96] Arthur J. Riel, *Object Oriented Design Heuristics*, Addison Wesley, 1996.

[Schoenfeld96] Ari Schoenfeld, *"Three Domain Specific Design Patterns"*, Proceedings of PloP '96, Track No. 7.

[SO97] Ari Schoenfeld y Hanna Oktaba. *"Personas y Roles", un patrón de diseño*, Encuentro Nacional de Computación 1997, Querétaro, México.

Notación

Se presenta un diagrama que utiliza la notación de Booch con los elementos utilizados en los diagramas del presente documento:



Del diagrama anterior se puede ver que:

Hay tres **clases** cuyos nombres son: Clase1, Clase2 y Clase3.

La Clase1 tiene un **atributo** (atributo2) y un **método** (operación2).

La Clase3 **hereda** de la clase2, y un objeto de la Clase2 **contiene** n objetos de la Clase1.

Three Domain Specific Design Patterns

Ari Schoenfeld Ba. Sc.
UACP y P del CCH
ari@servidor.unam.mx
Universidad Nacional Autónoma de México
México D.F.

Introduction

Patterns can be applied to any part of the software development process. Patterns deal mainly with Classes and relationships between them. There has been a lot of development and research in patterns, and we can see[II] *how there are patterns that focus on the analysis of the domain*[III], *others that focus on the design of software systems*[IV], *and some that focus on the technical issues in the implementation on a specific language*[V]. *This paper presents a proposal of patterns in the "problem-domain" level. These patterns deal with abstractions from the real world that are likely to appear in many software systems, specially in business an administration software systems. This article presents three basic patterns that can help to find a solution to some typical problems usually found in a wide range of business applications.*

Domain Specific patterns

One of the main advantages of using patterns is to *reuse* the experience of someone else to solve a problem. There are many kinds of patterns that can help in different phases of the software process (mainly Analysis, Design, and Implementation). Design patterns can deal with generic design issues applicable to almost any kind of software system. They usually don't deal with the mapping from the real world to a model. They mainly help us to find the best solution to a kind of design problems that could be found in any kind of system.

Domain Specific Design Patters are mostly useful in specific domain or "kind" of systems. On one hand, this kind of patterns are not useful for every software system (they are very specific). On the other hand, since these patterns are domain specific, they can help to find abstractions and kinds of relationships between classes that otherwise would take a long time and effort to find.

A typical problem in the first stages of design is to know if the classes we found and their hierarchy are right, that is, if it is going to work in the future, and if it is going to be resilient to change. The patterns we present have proven to solve the problems they are intended for, and also are quite stable and change-resistant. They were used in two very different systems, one is a textiles factory administration system, and the second one is an insurance management system for insurance agents. Three patterns are presented. The first one is the Conversions

pattern that helps to handle in a uniform way anything that can be converted from one scale to another, or from one rate to another. The second one, Persons-and-Roles is a solution to a problem that arises in every software system: An object can play many roles in a system and that could lead to a change in type, something that should be avoided if possible. Using a "role" class, this instance can play many roles without changing it's type. The third and most important, is the Documents-and-Roles pattern, which helps to handle documents and persons in a very flexible way, useful in many software systems.

These patterns are presented following the description format of Gamma et. al.[VI].

I.- Conversions Pattern

Intent:

Handle and save (for precision) quantities in their original format, but use them easily in any other representation or scale in a transparent way.

Motivation:

Where conversions are needed, this pattern is very helpful. Conversions can be made between currencies, or units of measure for example. The Conversions pattern describes what classes are needed in order to save data as it was originally generated so precision is saved but easily accessed in any other scale or converted with a different rate.

Applicability:

Use this pattern when there is a need to express data differently from how it was generated, maybe to be compared, related or added (for example) to other data. This, of course, can only happen if the data can be expressed in different units or "systems of measure" and can be translated from one to another.

For example, a company that has branches in different countries, would need to see it's financial results expressed in dollars to compare them, even if they were originally stored in pesos, yens, pounds, etc.

Also, the products the same company buys or sells could be originally registered in feet. Later on the *Company* may need to consolidate the quantities in meters or inches or whatever. With this pattern, you can avoid conversion mistakes, because the classes are responsible for making all the necessary calculations to give the correct result.

Structure:

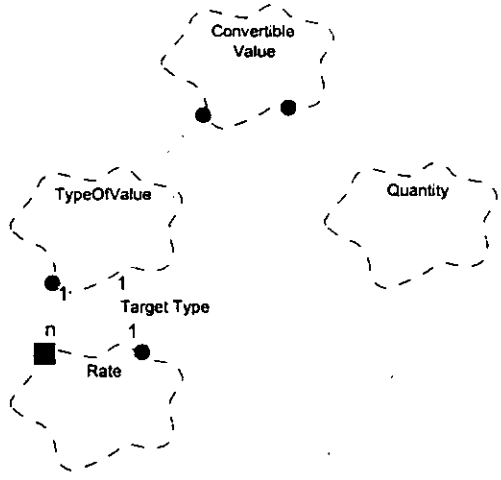


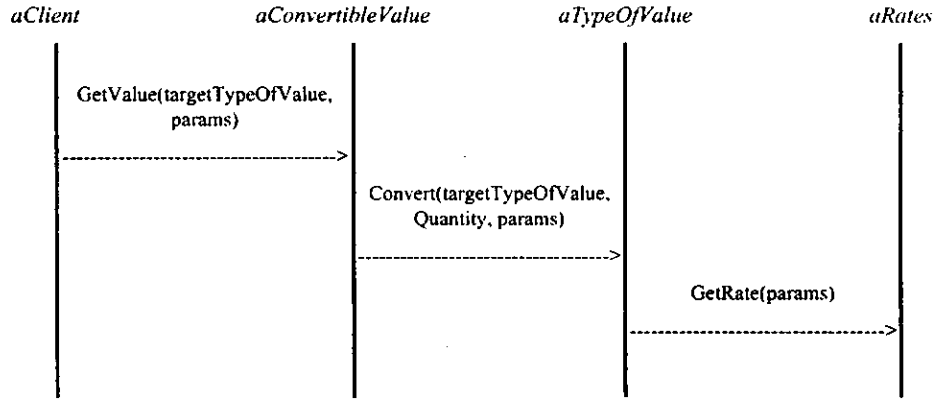
Fig. 1. - Structure of the Conversions Pattern.

Participants:

- *ConvertibleValue*: It is the class that has the value that might be converted. This is the class actually used in the system, and where the original value is stored (and will be saved just that way for later conversion, if needed).
- *Quantity*: Saves a numeric value, usually a real or alike.
- *TypeOfValue*: Saves the **type** of the value stored in *Quantity*. This class is the one responsible for calculating conversions between different *TypeOfValue*-s. The information needed to make all these conversions is stored in the *Rate* class.
- *Rate*: saves conversion information. One *TypeOfValue* can have conversion rates for all other *TypeOfValue* instances, or just for some of them.

Collaborations:

As we can see in the Object Interaction Diagram below, the *ConvertibleValue* class can be asked to display it's value in a different *TypeOfValue*, so it asks *TypeOfValue* to make a conversion which can be done thanks to the *Rate* class.



The *params* parameter can be used depending on the type of conversion. For example for currency conversions, it is necessary to define a date for the conversion. In other conversions, this could be simply not used.

ConvertibleValue is responsible for expressing it's quantity in any scale (if allowed). To make the conversion it relies on TypeOfValue which is the only responsible for making the conversions. To make the conversion, it uses the *Rate* class which has the data needed to calculate the desired conversion.

Consequences:

Data can be stored as it was originally generated, what preserves accuracy. Since the conversions are handled inside ConvertibleValue, data can be accessed as if it was of the same type, in the same scale so every query or report can be viewed in the desired scale (or according to the previous example, in the desired currency).

If in the future there is a new scale included in the system, all what is needed is to specify the rates for conversions, and "old" data can be expressed in this new scale (continuing with the previous example, this is specially useful in countries like Mexico where there are changes in the currency to "eat" zeroes (5000 pesos changed to be 5 "new" pesos). In this case there is no need to change even a line of code: simply use the "new" currency, and all the queries can be expressed in "old" or "new" pesos).

Implementation:

Quantity can be any numeric type.

As it was stated before, the parameters of the *GetValue* function may vary depending on the type of conversions.

Not all the possible rates for conversions might be captured in the system, so maybe some conversions will be impossible to calculate. In these cases it is necessary to define how to handle this situation (raising an exception, giving a default value, etc.).

There can also be illegal conversions (for example, feet can't be translated to degrees, or ounces to meters) and appropriate actions should be taken.

Most conversions are symmetric and this has to be assured (i.e. a value converted to a different scale and back, must be the same).

Known uses:

The most common and typical uses of this pattern are for Currencies (fig. 2), and Units of Measure (fig. 3). For example, the money class has a *Quantity* and a *Currency*. It must be very accurate in any calculation and conversion, but what's more important is that it is capable of expressing it's quantity in any currency at any given date (if the exchange rates are available). The conversions are actually calculated by the *Currency* class at a given date. Currency is responsible for saving and "knowing how" to convert between currencies at any given date.

We can send a message to Money like GetValue(currency2, date), that will give us the value in currency2 at a given date of the amount (quantity) this instance has.

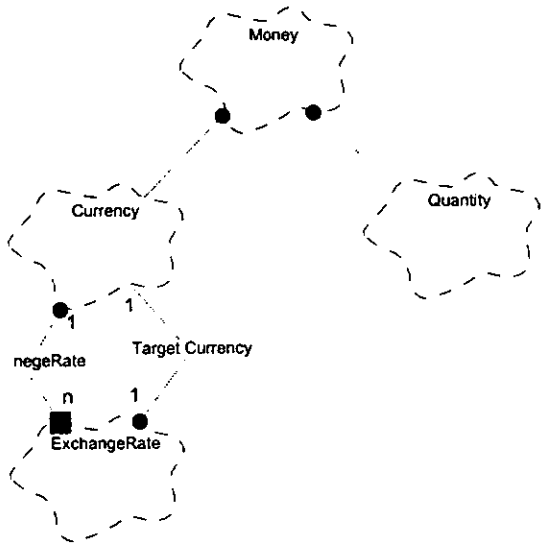


Fig. 2. -First example if the Conversions Pattern: Money.

This Pattern instantiated for Units of Measure could be applied in a stock tracking system, or any system dealing with products or merchandise. As we can see in Fig. 3,

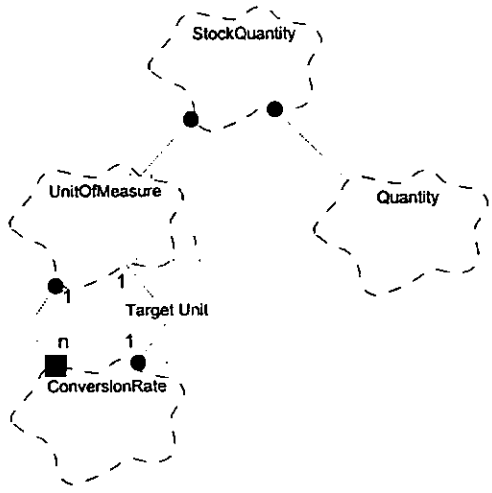


Fig. 3. - Second example of the Conversions Pattern: StockQuantity.

we could track a given textile (for example) so we have a StockQuantity instance that tells us that we have 1000 ft. We could send a message to this object GetValue('meters') and it would use UnitOfMeasure to calculate the corresponding meters. In this manner we can calculate the total stock in meters, feet or inches, as desired, even if part of the received merchandise was in feet, part in meters, and part in inches.

2.- Persons and Roles

Intent:

Persons in a system participate under different roles. Persons can freely change their roles at any time and this must not mean that a person object changes it's type. Also, a person can have many roles simultaneously and increase or decrease them as needed.

Motivation:

Whenever a software system deals with persons (could be users, employees, clients, students, companies, etc.) there is a very adaptive pattern that could be used in almost any of them, and solves a problem that has been discussed a lot. A *Person* can get a job, so he is an *Employee*, in the mid time he starts a degree in the university, so it's a *Student*, then since he is always sleeping at work they fire him so he is *Unemployed*, and so on. The problem here is that if we define a hierarchy where for example, *Employee* inherits from *Person*, just as *Student* or any other role (well yes, an *Employee is_a Person* and also a *Student* is_a *Person* so it's understandable why it's so common to model it like this), how can we express in our system that a *Person* is an *Employee*, and also a *Student*? The problem is that inheritance is a way of **classifying** that implies a partition, i.e. an instance of one type can't be at the same time one of the other types. Even a change of class (something every programmer want's to avoid) does not help because somebody *is_a Student* at the same time he *is_a Employee*.

What we see is that persons play many different **roles**, and most of the times, at the very same time. We need another class (*Role*) which will help us to model this. Being a *Student*, an

Employee, a *User*, a *Company*, are only *Roles* that the same *Person* could play at the same time. A *Person* has to be aware of all the roles it plays, and also a *Role* knows which *Person* it is about.

Applicability:

This pattern can be applied to any system that deals with persons and roles persons play. In almost any software system for an organization, persons and their roles are modeled in a way or another.

Structure:

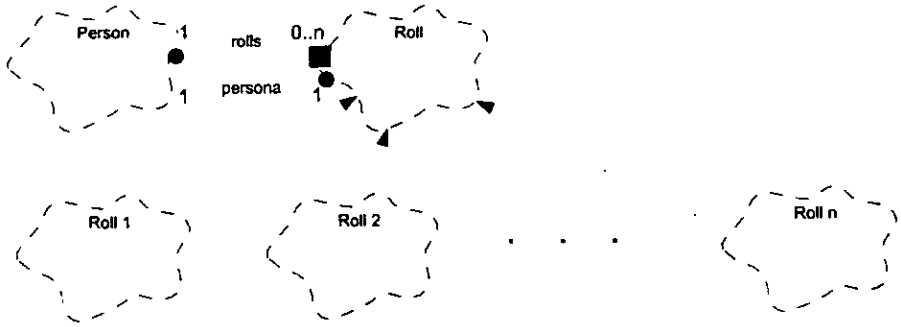


Fig. 4. -Structure of the Roles Pattern.

Participants:

- *Person* is the basis of all the possible roles in the system. A *Person* has all the data that does not change from role to role.
- *Role* is an abstract class from which any possible role in the system inherits. The specific roles have all the attributes related to that role.

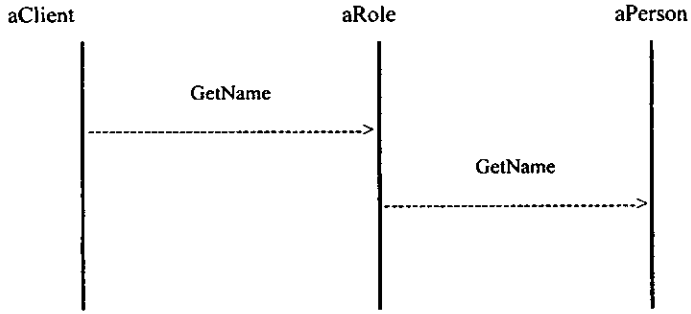
Collaborations:

A *Person* has a list of all it's *Role*-s. On the other hand a *Role* has a reference to the *Person* it is about. A *Role* could see the attributes that the *Person* owns as his, for example a role *Employee* could use the birth date (a *Person* attribute) to calculate the age for calculations or validations

Persons are normally used all around in any administration system. They have a list of attributes like name, date of birth, addresses, phones etc. Note that a Corporate Entity could also be seen as a *Person* (with the same attributes). With this pattern, *Role*-s are the ones to be used all around, and the *Person* will be accessed through the *Role*.

A *Role* actually is like an envelope for the class *Person* plus, it adds its own behavior. If the attribute needed is specific to the *Role*, the *Role* answers directly. If not, it asks *Person* for that attribute:

For example:



Consequences:

This pattern avoids the need of changing the type of an instance whenever is a change in the role a *Person* plays. We only need to create or destroy a *Role* object associated to that person.

All it's data like name, birth date, addresses, phones, etc. is accessible from all it's roles and could be seen as part of the roles themselves. Any *Role* could be created and destroyed as needed. A *Person* could even have two or more *Role*-s of the same kind at the same time (like two jobs, something not so strange nowadays).

Implementation:

There are some validations that should be made to avoid contradictory *Role*-s of a *Person*, like *Employee* and *Unemployed*, or *Mother* when de attribute sex of the *Person* is male. The *Role*-s are responsible for making these validations whenever they are associated with the corresponding person. Also there are *Role*-s only for individual persons, only for Corporate Entities, or for both.

In the system, almost all the references should be to *Role*, and not to *Person*. The *Person* can be accessed via the *Role*.

Known Uses:

In the following diagram we can see some examples of *Role*-s.

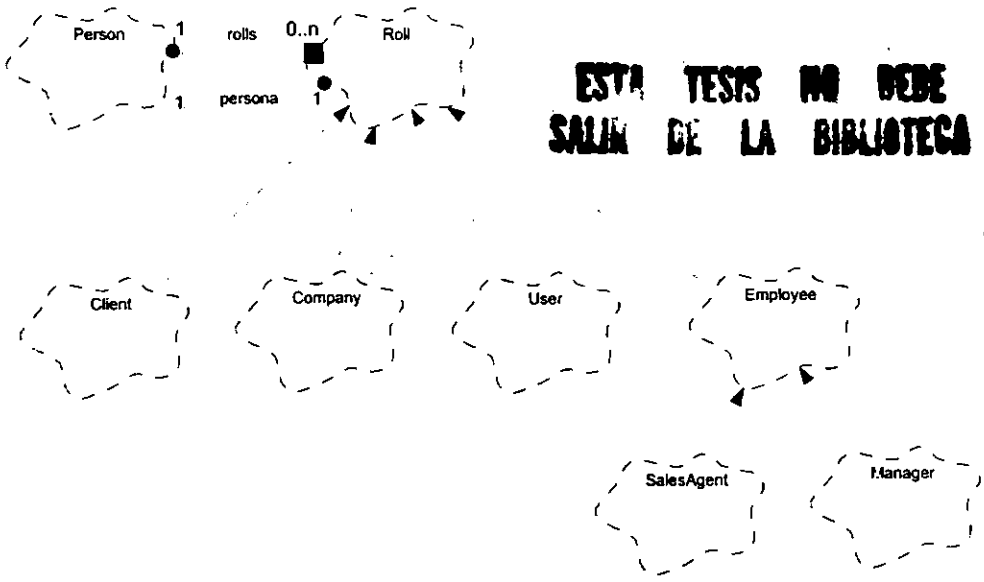


Fig. 5. - Example of the Persons and Roles Pattern.

This pattern was used in two systems: one for a textiles factory and one for an insurance management system for insurance agents. In the first one, the company decided to sell at special prices some of the products it manufactured to their employees. Since all the employees were already in the system, all what was needed was to create the *Client* roles for them and capture a few attributes for this *Role*.

In the second one, a *Person* in the system can become *Insured* and if he/she pays the insurance, it will also be a *Client*, and this can be applied anytime to anyone in the system, something very practical when you talk about many persons that change roles very often.

Related Patterns:

Coplien's **Envelope-Letter** idiom is a technique for changing an object's class at run-time. This is not exactly what we are doing here since we don't only want to change the class of an object, we also want to approach it in different ways (even many of the same kind) at the same time.

The **State** pattern in Gamma's book also makes it possible for an object to appear to change it's class, but here too, the object changes it's class but only one at the time.

Note that this pattern could actually be generalized so any entity could have many roles.

The OOram Method has a deep coverage of roles of entities but with a different approach[VII].

3.- Documents and Roles

Intent:

Express any relationship of documents and roles of persons in a system.

Motivation:

In every organization, most of the work that has to be tracked, managed, followed up, saved for future queries, etc., deals with documents, and people. Actually, documents are always related to people, and people to documents.

This pattern makes the expression and representation of all the possible relationships between documents and roles very flexible. It also makes it very easy to change whenever needed.

This pattern enables us to attach any number of documents to a document, any number of *Role*-s to a *Role* (of *Person-s*), and any number of documents to a *Role* of a *Person* (and vice versa).

Applicability:

Use this pattern in any administrative system when you need flexibility and resilience to change in the relationships of *Document-Document*, *Role-Role* (of *Person*-s), and *Document-Roles*.

Structure:

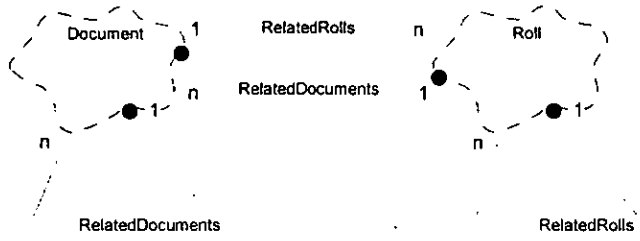


Fig. 6. - Roles and Documents Pattern.

Participants:

- *Documents* are very common in administrative systems. Every document in the system should inherit from *Document*.

This class (and it's descendants) map every real-life document to the system. Every administration system deals with lot's of classes of documents.

- *Role* is a role of a *Person* and is explained in the Persons and Roles Pattern.

Collaborations:

The structure is what is important in this pattern. There are two methods for both *Document* and *Role*: GetDocument and GetRole. This two methods look for the roles and documents in their respective lists, so they can be fetched.

The collaboration between classes depends on the actual classes and the specific scenarios they are involved in. For what this pattern is concerned, there will be multiple calls to GetDocument or GetRole to get the related instances and to AddRole or AddDocument to include *Document*-s and *Role*-s to the lists.

With this structure, it is possible to navigate through the documents that are related and trough the people involved. For example, if there is a problem with a given *Client*, we can look at all the documents related to him, and lets say we find the one with the problem. From that document we can find who contacted the client, what letters were sent, who transported the goods, who received them in the client's side, and everything to find out what caused the

problem and what *Person* we have to talk to. The collaborations here are between all this related *Documents* and *Roles* through their links.

Consequences:

This pattern allows much flexibility, adaptability an resilience to change of any relationship of *Document-Document*, *Role-Role* (of *Person*-s), and *Document-Role*s. Most of the time changes in an organization relate to changes in these relationships. In this case, this pattern will save us from making deep changes in the classes, we will only need to allow, disallow or require the addition of another kind of relationship.

Implementation:

It's important to validate the allowed relationships.

Whenever a *Document* is inserted in the *Role* RelatedDocuments list, this *Role* should appear automatically in the RelatedRoles list of the *Document*.

Note that the *Role*-s themselves describe the type of relationship. For example, a *Client* role related to an *Invoice* tells us that he is the one the *Invoice* is made for. A *Client* related to a check, tells us who made the check. The same *Person* after this *Client* with the *Provider* role related to a check, tells us who this check we made it for. So, as we can see, the meaning of the relationship between a *Role* and a *Document* is in the type of role.

Known Uses:

For example, a *Letter is_a Document*, that has a *Sender*, and a *Addressee*, which are *Person*-s (*Role*s of *Person*). On the other hand, these two people have this *Document* related to them (for the first one as a *Letter* that he sent and the second one as a *Letter* he received). Any *Letter* related to a *Sender* role is a *Letter* this *Person* wrote. Just as any *Letter* related to an *Addressee* role is a *Letter* this *Person* received.

The same happens for example with an *Invoice*. The people (or *Role*s) involved are: The *Company* that issues the *Invoice, the Person* or *Company* the *Invoice* is made for (*Client*), the *Person-s* (*Employees*) who made the sale (*Agents*, *Customer_Support*, etc.), the *Person* that has to be called to know when the check will be ready, and any other *Person* that could become related, can be easily added with this pattern. Notice that the *Role* itself describes the relationship with the *Document* (i.e. the *Role Agent* related to an invoice tells us who was the agent for that sale). Also, a document has a list of it's related documents. From the last example, the *Invoice* could have the *Check* that paid it, the *Shipment* it was sent with, the client-satisfaction *Questionnaire*, or if it was canceled, the *Invoice* that replaced it.

Conclusion

Although Object Oriented Programming solves many problems of Procedural Programming, there is still a complexity in software systems that shouldn't be overlooked. Even small Object Oriented Systems have more objects than can be handled by a (normal) human mind. The use of patterns can so greatly reduce the complexity of the design and implementation, that they should be used whenever possible. Moreover, they can help us (and they did) to achieve a great deal of reuse.

II Oktaba Hanna, "Patrones: lo que nos hacía falta para aprovechar el modelo de Objetos" ...

III P Coad, D. North & M. Mayfield, "Object Models Strategies, Patterns and Applications", Yourdon Press, Prentice Hall, 1995.

IV Gamma Erich, ... "Design Patterns - Elements of Reusable Object Oriented Software", Addison Wesley, 1995.

V J.O. Coplien "Generative Pattern Languages: An emerging direction of software design", C++ Report, Jul-Aug 1994.

VI Gamma Erich, ... "Design Patterns - Elements of Reusable Object Oriented Software", Addison Wesley, 1995, pp 6-7.

VII Reenskaug Trygve, P. Wold and O.A. Lehne, "The OOram Software Engineering Method",