Switch Strategy Pattern

Luis César Maiarú[1]
Universidad Argentina de la Empresa (UADE)
Lima 77, (073) Buenos Aires, Argentina
luiscesar@argentina.com, lmaiaru@uade.edu.ar

Abstract

The Switch Strategy pattern allows to select different implementations of the same interface, depending on determined condition, without hard-coding a switch statement or a sequence of conditional statements.

1 Intención

Encapsular un conjunto de opciones compartiendo la misma interfaz. Esto permite a los clientes seleccionar una opción sin conocer la política de selección.

2 Otro Nombre

Case Strategy.

3 Motivación

Suponga que se quiera saber el precio que paga cada registro en un congreso. El mismo depende de la categoría del participante, por ejemplo, Profesor, Estudiante, Profesional miembro de una asociación reconocida, o ninguna de las opciones anteriores.

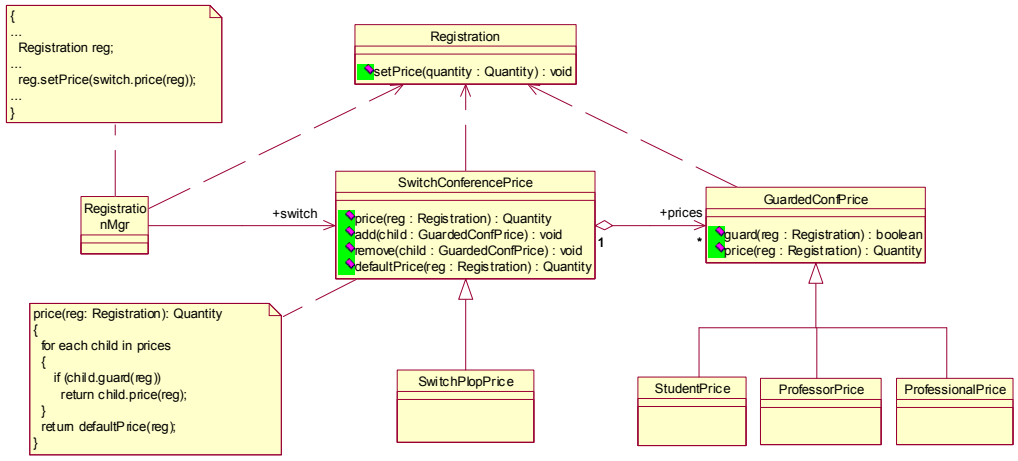
Supongamos que el precio ha ser pagado depende de la categoría del participante y otra información incluida en el registro.

Nos encontramos con la situación de seleccionar entre varias opciones, las cuáles comparten el mismo tipo de entrada y el mismo tipo de salida, pero dependiendo de cada opción una estrategia diferente será usada.

El problema es cómo diseñar una solución que satisfaga los siguientes requerimientos:

- Si las categorías o el porcentaje asociado a las mismas cambian no haya que re-escribir el código del cliente, es decir que se puedan agregar nuevas categorías en modo dinámico.
- Que el cliente no tenga la responsabilidad de tratar con el orden de evaluación si un participante pertenece a mas de una categoría.
- Que el diseño se pueda re-usar en diferentes congresos.

La solución obtenida es manejar estrategias custodiadas (guarded strategies). Un administrador de estrategias custodiadas (SwitchStrategy).) será el responsable de administrar el orden de las estrategias. La primera cuya custodia satisfaga la condición de evaluación será la elegida. El cliente tratará solo con dicho administrador. Además, el administrador tendrá las responsabilidades de agregar o sacar estrategias custodiadas. El administrador poseerá también una estrategia por defecto en caso que de que ninguna de las custodias hayan sido satisfechas.



La clase RegistrationMgr será la responsable de asignarle un precio a cada registro. Para ello se basará en la clase SwitchConferencePrice la cuál tiene una lista de estrategias custodiadas cuyas clases implementan la interfaz GuardedConferencePrice. Estas clases implementan diferentes custodias y estrategias:

GuardedProfessorPrice implementa la custodia para los participantes que son profesores e implementa la estrategia que determina el precio del registro para el mismo dependiendo del descuento.

GuardedStudentPrice implementa la custodia para los participantes que son estudiantes e implementa la estrategia que determina el precio del registro para el mismo dependiendo del descuento.

GuardedProfessionalPrice implementa la custodia para los participantes que son profesionales acreditados e implementa la estrategia que determina el precio del registro para el mismo dependiendo del descuento.

La clase Quantity ha sido usada para representar el tipo del precio en reemplazo de tipos mas restrictivos como double o Double con el fin de independizarnos de cuestiones implementativas.

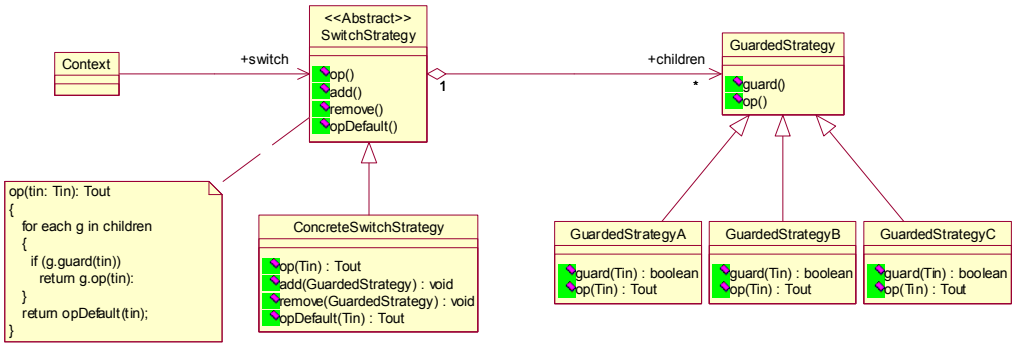
4 Aplicación

Use el patrón Switch Strategy en los siguientes casos:

- Es necesario elegir una opción de un conjunto de bloques de instrucciones dependiendo de una determinada condición. Cada bloque de instrucciones puede ser encapsulado en una operación.

- Es deseable que los clientes no tengan la responsabilidad de preguntar por la condición, sino solo llamar a una operación (interfaz), cuya implementación será seleccionada entre varias dependiendo de una condición.
- Es deseable que los clientes no manejen el orden en que las distintas opciones serán controladas por si aplican o no.
- Es deseable poder agregar o eliminar opciones en modo dinámico.

5 Estructura



6 Participantes

- **SwitchStrategy**

1. Declara la interfaz para la operación que los clientes van a usar.
2. Busca entre todos sus hijos, siguiendo un orden predeterminado, por el primero que satisfacerla condición, en tal caso invoca su estrategia.
3. Almacena los objetos hijos del tipo GuardedStrategy.
4. Declara la interfaz para una operación por defecto en caso en que ninguno objeto hijo satisfaga la condición.

- **ConcreteSwitchStrategy**

1. Puede redefinir el orden de búsqueda entre todos sus hijos.
2. Implementa la operación de defecto.
3. Implementa el agregado y la eliminación de los hijos (opciones).

- **GuardedStrategy**

1. Declara la interfaz de la operación y de la guarda.
2. Representa una opción a ser seleccionada.

- **GuardedStrategyA**

1. Implementa la interfaz de la operación, que es el objetivo del cliente.
2. Implementa la interfaz de la guarda, la cuál es usada para elegir entre las distintas opciones.

- **Context**

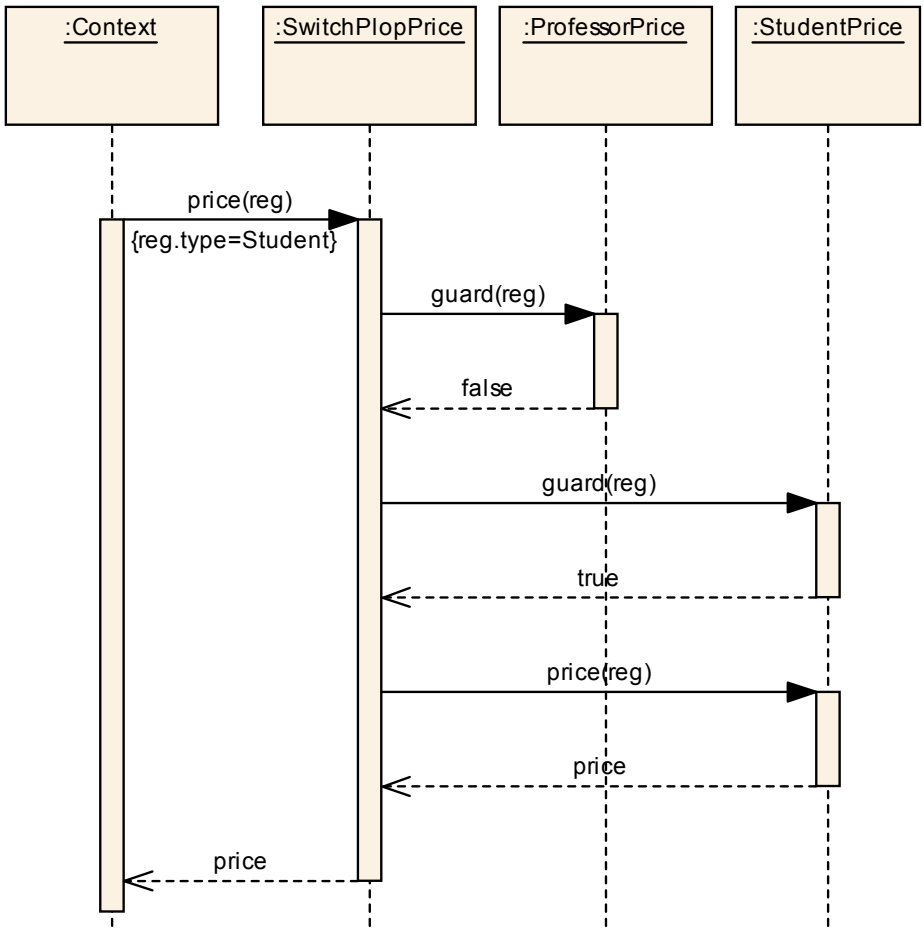
1. Mantiene una referencia a una instancia de SwitchStrategy, donde la selección y las opciones de las posibles implementaciones han sido encapsuladas.
2. Mantiene una referencia a un objeto SwitchStrategy, donde las opciones y la selección están encapsuladas.

- **Tin**

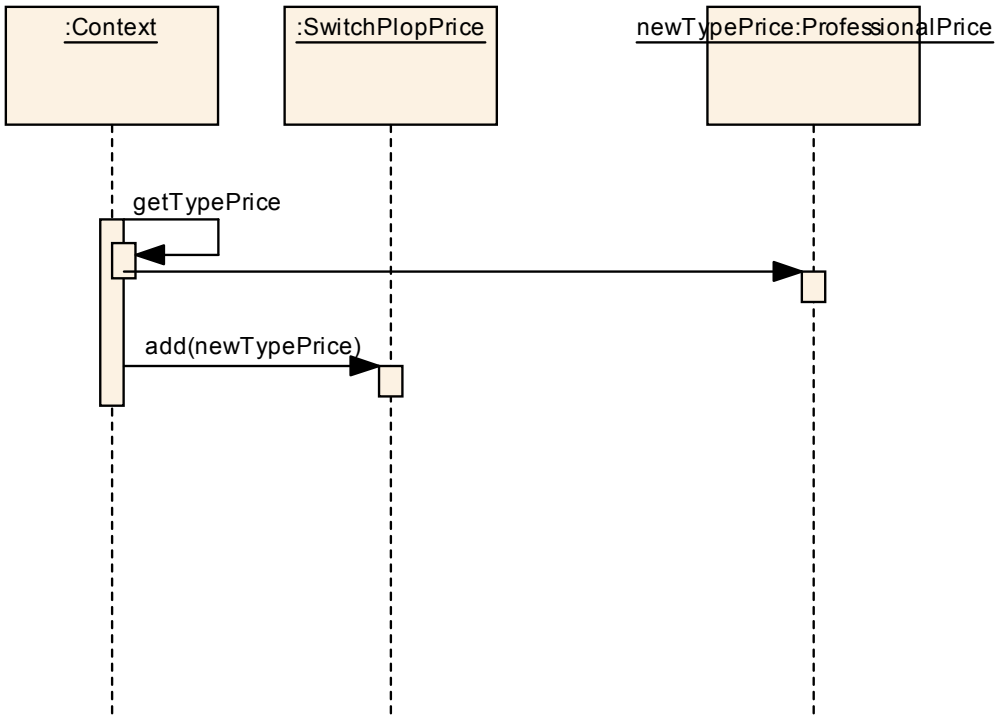
1. Es el tipo del objeto de input usado para verificar la condición de selección (custodia) y para determinar el resultado (objeto de tipo **Tout**).

7 Colaboraciones

- Los clientes usan la clase `SwitchStrategy` para referirse a la estrategia en modo uniforme. La selección de las distintas estrategias está encapsulada en la clase `SwitchStrategy`. La estrategia seleccionada será la *primera* que satisfaga la custodia (guard). Si la condición no es satisfecha por ninguna de las opciones entonces una operación por defecto es llamada.



- Las estrategias *custodiadas* pueden ser agregadas o eliminadas al objeto `SwitchStrategy` en modo dinámico sin tener que modificar el código del cliente que usa la operación.



8 Consecuencias

El patrón Switch Strategy tiene las siguientes ventajas:

1. *Facilidad en el agregado de una nueva opción en modo dinámico.* El requerimiento de considerar una opción de estrategia significaría agregar un apropiado hijo al objeto `SwitchStrategy`.
2. *Flexibilidad para determinar el orden de selección.* El orden de selección de las distintas opciones viene dado por la implementación de la subclase de `SwitchStrategy`, `ConcreteSwitchStrategy`.
3. *Flexibilidad para determinar la custodia.* La custodia (guard) de que cada estrategia es así mismo una estrategia, cuya implementación puede variar libremente respetando su interfaz con el contexto a través de la clase `Tin`.
4. *Los clientes no se deben preocupar por las diferentes estrategias.* A diferencia del patrón Strategy [1] donde el cliente se debe preocupar por cada una de las implementaciones de las estrategias, en este caso, el cliente solo se preocupa por la interfaz dada por la clase `SwitchStrategy`.

El patrón Switch Strategy tiene las siguientes desventajas:

1. *Sobrecarga en la comunicación entre el Contexto y el SwitchStrategy.* Similar al patrón Strategy [1] la interfaz `GuardedStrategy` es compartida por todas las clases `ConcreteGuardedStrategy` sin importar si los algoritmos son simples o complejos. Además en este caso la interfaz está referida a dos operaciones: la custodia y la estrategia. Así, puede ser probable que algunas de las clases `ConcreteGuardedStrategy` no usen toda la información brindada.
2. *Incremento del número de objetos.* También similar al patrón Strategy se incrementa el numero de objetos en las aplicaciones.

3. *Sobrecarga de ejecución en casos de estrategias conocidas.* Cuando las distintas posibles estrategias no cambian muy a menudo, entonces el cliente podría invocar directamente a las respectivas implementaciones. En este caso el patrón Strategy sería usado y se evitaría la búsqueda por la custodia satisfecha.

9 Implementación

1. *Definiendo la interfaz entre Context, SwitchStrategy y GuardedSrategy.* El paso de la información entre el `Context`, la clase que implementa `SwitchStrategy` y las que implementan la interfaz `GuardedStrategy` debería realizarse en un modo eficiente. En el diseño hemos propuesto hacerlo a través de parámetros (la clase `Tin`). Eso tiene la ventaja de hacer la vinculación en tiempo de ejecución. Sin embargo, como la información contenida en la clase `Tin` es la necesaria para cubrir todos las opciones podría haber casos en la evaluación de la custodia o de la operación que no sea utilizada toda esa información. Una posibilidad para evitar estos casos es la utilizar clases filtros en la `ConcreteSwitchStrategy`, para determinadas `GuardedStrategy`.
2. *Haciendo la estrategia por defecto es opcional.* La estrategia por defecto, que viene usada solo en los casos que ninguna custodia ha sido satisfecha es opcional. Podrían haber casos en donde el conjunto de objetos custodiados asegure la completitud de las evaluaciones.

10 Ejemplo de Código

Daremos una presentación de código (nivel medio) Java para el ejemplo presentado en la sección Motivación.

La clase `RegistrationMgr` tendrá la responsabilidad de responder mensajes que hagan referencia a los registros.

```
public class RegistrationMgr
{
      // …
      public Quantity price(Registration reg)
     {
            SwitchConferencePrice switch = new SwitchPlopPrice();
            return switch.price(reg);
     }

} // class RegistrationMgr
```

Cada registro contará con al menos dos atributos: la categoría del participante y el porcentaje de descuento a ser sustraído del precio total. También ha sido incluido el atributo `basePrice` representando el precio base de cada registro.

```
public class Registration
{
      Quantity price;
      String category;
      Quantity basePrice;
```

```
	public void setPrice(Quantity priceC)
	{
		price = priceC;
	} // setPrice(Quantity): void

	public Quantity getBasePrice()
	{
		return basePrice;
	} // getBasePrice(): Quantity

	// …

} // class Registration
```

La clase `Quantity` representa el tipo asociado al precio.

```
public class Quantity
{
	double value;

	public Quantity(double valueC)
	{
		value = valueC;
	} // constructor(double)

	public double getValue()
	{
		return value;
	} // getValue(): double

	// …
} // class Quantity
```

La clase abstract `SwitchConferencePrice` define el tipo de información que será usada por la implementación `SwitchPlopPrice` y por los objetos `GuardedConferencePrice` ligados a la misma.

```
public abstract class SwitchCoferencePrice
{
	Collection children;

	public Quantity price(Registration reg)
	{
		GuardedConferencePrice gCP;
		Iterator it = children.iterator();
		while (it.hasNext())
		{
			gCP= (GuardedConferencePrice) it.next();
			if (gCP.guard(reg))
			{
				return gCP.price(reg);
			}
		}
		return defaultPrice(reg);
```

```
	} // price(Registration): Quantity

	public void add(GuardedConferencePrice) {}
	public void remove(GuardedConferencePrice) {}
	public Quantity defaultPrice() {}

} // class SwitchConferencePrice
```

La clase `SwitchPlopPrice` extiende la clase abstract expuesta más arriba. En este caso, la forma en recorrer la lista de objetos `GuardedConferencePrice` no es considerada en detalle. La primera que valida la custodia es la seleccionada.
Además se implementa `defaultPrice` retornando un valor dependiendo del descuento que posea el registro.

```
public class SwitchPlopPrice extends SwitchConferencePrice
{
	public SwitchPlopPrice()
	{
		children = new Vector();
	}

	public void add(GuardedConferencePrice gCP)
	{
		children.addElement(gCP);
	} // add(GuardedConferencePrice): void

	public void remove(GuardedConferencePrice gCP)
	{
		children.removeElement(gCP);
	} // remove(GuardedConferencePrice): void

	public Quantity defaultPrice(Registration reg)
	{
		// Return a default value
		return new Quantity(5 * reg.getBasePrice.getValue());
	} // price(Registration): Quantity

} // class SwitchPlopPrice
```

La interfaz `GuardedConferencePrice` define la forma en que el pasaje de información es realizado, tanto para la custodia como para la operación precio. En este caso, el mismo tipo de parámetro es usado.

```
public interface GuardedConferencePrice
{
	public boolean guard(Registration reg);
	public Quantity price(Registration reg);
} // interface GuardedConferencePrice
```

Las clases `GuardedProfessorPrice`, `GuardedStudentPrice` y `GuardedProfessionalPrice` implementan la interfaz descripta anteriormente. Cada una de estas clases tiene su propia interpretación de la custodia y del precio del respectivo

registro. El cuál, a igual que en el caso por defecto, depende del registro pasado como parámetro.

```
public class GuardedProfessorPrice
      implements GuardedConferencePrice
{
      public boolean guard(Registration reg)
      {
            return reg.category.equals(new String("Professor"));
      }
      public Quantity price(Registration reg)
      {
            return new Quantity(3 * reg.getBasePrice.getValue());
      }
} // class GuardedProfessorPrice

public class GuardedStudentPrice
      implements GuardedConferencePrice
{
      public boolean guard(Registration reg)
      {
            return reg.category.equals(new String("Student"));
      }
      public Quantity price (Registration reg)
      {
            return new Quantity(2 * reg.getBasePrice.getValue());
      }
} // class GuardedStudentPrice

public class GuardedProfessionalPrice
      implements GuardedConferencePrice
{
      public boolean guard(Registration reg)
      {
            return reg.category.equals(new String("Professional"));
      }
      public Quantity price(Registration reg)
      [2]{
            return new Quantity(4 * reg.getBasePrice.getValue());
      }
} // class GuardedProfessionalPrice
```

11 Usos Conocidos

- Un particular uso de este patrón es en la validación de reglas las cuáles serán evaluadas dependiendo de determinadas condiciones. Por ejemplo, el Switch Strategy fue usado en el *validador* de ofertas de un Sistema de Mercado Electrónico de Bolsa de Valores[2]. Al ser ingresada una oferta (de compra o de venta) por los operadores, se debía verificar su validez antes de que la misma pudiese ser utilizada para concertar operaciones. La oferta tenía distintos atributos. Verificar la validez de la oferta implicaba evaluar distintas reglas dependiendo de condiciones que hacían referencia a

[2] Electronic Stock Exchange Market System

dichos atributos. Debido a las fuertes restricciones de tiempo en el ingreso de una oferta, verificar todas las reglas aún aquellas que no aplicasen no era deseado. El Switch Strategy permitió agrupar bloques de reglas cuya validación dependían de una misma condición. El *validador* de ofertas solo tenía que interactuar con bloques *custodiados* de reglas.

- Otro uso conocido de este patrón fue en la programación de un *Broker* Java para Base de Datos. El*Broker* hacía la veces de intermediario entre el cliente y la base de datos, abstrayendo al primero de interactuar directamente con los distintos conceptos relacionados con la base de datos. Una de las responsabilidades del Broker era la de *asociar* clases Java con tipos de Base de Datos en modo *completamente arbitrario*. El Switch Strategy permitía que la *estrategia de asociación* tuviera siempre la misma interfaz, haciendo la custodia (*guard*) referencia a las distintas clases y tipos.
- El patrón Switch Strategy fue también utilizado en un Sistema de Categorización de Especies (Stock). Dada una lista de especies, las mismas, de acuerdo a sus atributos, eran agrupadas por categorías. Por cada categoría luego se calculaban distintos tipos de información como aranceles y totales. Las categorías y las condiciones de pertenencia de una especie a una categoría podían variar frecuentemente. El Switch Strategy permitía obtener en todo momento la categoría a la que pertenecía una especie sin modificar el código del cliente. Así mismo permitía asociar y desasociar especies de las categoría en modo dinámico.

12 Patrones Relacionados

Flyweight [1]. El patrón Switch Strategy tiene una directa relación con el patrón Flyweight. La clase Tin podría ser usada como la clave (key) en el patrón Flyweight. La diferencia radica en que en el caso del Switch Strategy no se necesita una relación uno a uno entre las claves y las estrategias. El orden de selección viene determinado por la clase ConcreteSwitchStrategy. Además la creación de la estrategia custodiada es independiente de la clave, cosa que si ocurre en el patrón Flyweight.

Composite [1]. El patrón Switch Strategy se complementa muy bien con el patrón Composite. Usándolos en conjunto es posible anidar bloques custodiados de instrucciones.

Strategy and Abstract Factory [1]. El patrón Switch Strategy puede ser también un buen complemento de los patrones Strategy y Abstract Factory. Cuando el patrón Strategy es usado por un cliente, una estrategia concreta deberia ser usada. Esto puede ser realizado usando el patrón Abstract Factory. El Switch Strategy podría ayudar al Abstract Factory en el proceso de selección si varias estrategias están siendo consideradas.

The Adaptive Object Model Architectural Style [6]. El patrón Switch Strategy puede ser usado con el *Adaptive Object Model Architectural Style.* El Switch Strategy podría ayudarlo en el manejo de grupos de reglas de negocio cuya validación dependa de una determinada condición. También, es posible usar el Switch Strategy en reemplazo del patrón Strategy cuando en tiempo de ejecución es requerido considerar una familia de algoritmos simultáneamente.

13 Agradecimientos

El autor quisiera agradecer al *shepherd* Jorge L. Ortega Arjona por sus valiosos y precisos comentarios. Los mismos constituyeron un importante aporte para el mejoramiento

del presente trabajo. Además quisiera agradecer a los participantes del grupo en el congreso, quienes brindaron positivas sugerencias.

14 Referencias

[1] Gamma, E., Helm, R., Johnson R., and Vlissides, O. (995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison Wesley Longman.
[2] Fred Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
[3] Wolfgang Keller, *Mapping Objects to Tables, A Pattern Language.* In Proceedings of the European Pattern Languages of Programming Conference, Irrsee, Germany, 1997. Also at http://www.objectarchitects.de.
[4] Wolfgang Keller, *Object/Relational Access Layers, A Roadmap, Missing Links and More Patterns.* In Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing, 1998. Also in http://www.objectarchitects.de.
[5] Floyd Marinescu, *EJB Design Patterns, Advanced Patterns, Process and Idioms,* John Wiley and Sons, 2002.
[6] Joseph W. Yoder and Ralf Johnson, *The Adaptive Object Model Architectural Style,* IEEE/IFIP Conference on Software Architecture 2002 (WICSA '02) at the World Computer Congress in Montreal 2002, August 2002.