Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

Laboratorio de Programación - Trabajo Práctico *Programación Orientada a Aspectos*

La programación orientada a aspectos (POA) propone una extensión a la programación orientada a objetos, pensada para resolver el problema de las incumbencias cruzadas. ¿Qué son las incumbencias cruzadas?
Son trozos de código que se encuentran dispersos en varios lugares de una aplicación cruzándola transversalmente.
La POA propone separar ese código disperso, en una clase especial a la que llamamos "aspecto".
Luego los aspectos son entretejidos con las clases de la aplicación para generar el producto final.

Intervienen los siguientes conceptos:
join point (punto de unión, en adelante JP), es un **concepto** que se refiere al lugar en el código de la clase en que se debe entretejer un aspecto.

pointcut, (en adelante PC), es un enunciado que indica un conjunto de join point a considerar. Un point cut puede ser con nombre o anónimo, y para su definición se pueden utilizar conectivos lógicos (&&, ||, !). Es una cosntrucción que selecciona join points y colecta información del contexto, por ejemplo si se trata de la llamada a un método, la información de contexto que se puede capturar es el objeto *target* sobre el cual el método fue llamado, y los argumentos del método.

advise, es un trozo de código en el que se indica en que momento hay que considerar los pointcut, y que acciones hay que llevar a cabo. Un advise puede ejecutarse *before* (antes), *after* (después) o *around* (durante) el join point

Pointcuts + advice = reglas de entretejido dinámico. Los pointcuts identifican los join points requeridos, y los advice proveen las acciones que ocurrirán en los join points.

Sintáxis de un pointcut:
[id acceso] *pointcut* nombre([..]): **tipo-pointcut (signatura)**

Categoría de JP /	**sintáxis del PC**
Method execution	execution(MethodSignature)
Method call	call(MethodSignature)
Constructor execution	execution(ConstructorSignature)
Constructor call	call(ConstructorSignature)
Class initialization	staticinitialization(TypeSignature)
Field read access	get(FieldSignature)
Field write access	set(FieldSignature)
Exception handler	execution handler(TypeSignature)
Object initialization	initialization(ConstructorSignature)
…	…

Aspecto: es la unidad central de AspectJ. Contiene el codigo que expresa las reglas de entretejido (weaving). Los aspectos contienen pointcuts, advises, introducciones, declaraciones, y tambien datos,

métodos, etc, como una clase normal.
Una introducción es estática y sirve para introducir cambios a los elementos del sistema.
Una declaración es estática y permite agregar warnings y errores en tiempo de compilación.

Al trabajar con aspectos y diseñar el comportamiento transversal, el primer paso es identificar los JP en los cuales se pretende aumentar o modificar el comportamiento. Paso seguido se diseña cual será el nuevo comportamiento. Dentro del aspecto se escriben los PC que capturan los JP de interés, y luego se crean los advise para cada PC y se codifica en él la acción que se requiere se ejecute cuando se alcance el JP.

Ejercicio 1
Escriba el siguiente programa en java

```
public class HolaMundo {
   public static void main (String[] args){


       MensajeUno   menUno = new MensajeUno("lunes");
       MensajeDos   menDos = new MensajeDos("viento", 3);

       menUno.setMensaje(" de primavera");
       menDos.setMensaje("y sol");
       mensaje();
       mensaje(menUno);
       mensaje(menDos);
   }

   public static void mensaje(){
       System.out.println ("hola mundo");
   }
}
```

```
public interface Mensaje {

   public String getMensaje();
   public void setMensaje(String mens);
}
```

```
public class MensajeUno implements Mensaje {

   private String uno;

   public MensajeUno (String cad){
              uno = cad;
```

```
    }


    public String  getMensaje(){
        return uno;
    }


    public void setMensaje(String otro){
        uno = uno + "   " + otro;
    }
}



public class MensajeDos implements Mensaje {
    private String dos;
    private int    repeticiones;

    public MensajeDos (String cad, int rep){
          dos = cad;
          repeticiones = rep;
    }

    public String  getMensaje(){
        return dos;
    }

    public void setMensaje(String otro){
        String aux = "***" + dos + "****" + otro;
        dos = aux;
        for (int i=1; i<= repeticiones; i++)
              dos = dos + "\n" + aux;
    }

}
```

Al ejecutarlo se obtendrá la salida esperada, o sea el mensaje

```
hola mundo
lunes   de primavera
***viento****y sol
***viento****y sol
***viento****y sol
```

Ahora vamos a incorporar el uso de aspectos para agregar comportamiento dinámicamente, y sin necesidad de modificar el código del main, ni de las clases *MensajeUno* y *MensajeDos*

Paso 1:
crear el aspecto Saludos como sigue

```
public aspect Saludos {

   pointcut saludar() :
        execution (void HolaMundo.mensaje());

   after(): saludar(){
         System.out.println(" JAJAJA ");
   }

   before(): execution (void HolaMundo.mensaje()){
         System.out.println("Hi!!!! ");
   }
}
```

Declaración de un PC al que llamamos saludar(), que no tiene parámetros

Un advise que considera el PC saludar()

Un advise que contiene la declaracion del PC, es un PC anónimo

Y volver a ejecutar HolaMundo, como aplicación de AspectJ. Ahora se obtendrá la salida siguiente:

```
Hi!!!!
hola mundo
 JAJAJA
lunes   de primavera
***viento****y sol
***viento****y sol
***viento****y sol
```

Es decir a la salida anterior se agregaron 2 líneas, “HI!!!!” y “JAJAJA”:

- “HI!!!!” proviene del “advise before”, es decir se ejecutó “antes” de la ejecución del método *mensaje()* de HolaMundo

- “JAJAJA” proviene del “advise after”, es decir se ejecutó “después” de la ejecución del método *mensaje()* de HolaMundo

El PC *saludar()* indica que el JP que interesa considerar (es decir los puntos en el código en que se desea entretejer el aspecto) es cada lugar en que se produzca la ejecución de un método llamado *mensaje(),* sin parámetros, que corresponda a la clase HolaMundo, con un resultado de tipo *void.*

Si observamos la pestaña “Cross References” veremos lo siguiente:

```
HolaMundo
    mensaje()
```

```
advised by
 Saludos.after(): saludar..
 Saludos.before(): <anonymous pointcut>
```

Esto indica que al ejecutar el "HolaMundo" se detecto la existencia de un aspecto y se hizo el "weaving" dinámico, o sea el "entretejido" del aspecto en la aplicación HolaMundo. Concretamente dice que al ejecutar el "HolaMundo" se aplicaron 2 "advise":

- uno "*after* (despues)" según el pointcut "*saludar*"
- uno "*before* (antes)" según un pointcut anónimo

Esto es exactamente lo que le indicamos en el aspecto.

Ejercicio 2
Modifiquemos el aspecto ahora para hacerlo mas general. Para ello se pueden utilizar wildcards.

```
public aspect Saludos {
    pointcut saludar() :
      execution (void *.setMensaje(String));


      after(): saludar(){
            System.out.println(" JAJAJA");
      }

      before(): execution (void HolaMundo.mensaje()){
            System.out.println("Hi!!!! ");
      }
}
```

Ahora el PC *saludar()* se define para considerar la ejecución de los métodos *setMensaje(String)* correspondientes a **cualquier clase**, siempre que reciban un parámetro de tipo String. Quiere decir que la instrucción `System.out.println(" JAJAJA")` se ejecutará "after" de la ejecución del método *setMensaje* de la clase MensajeUno y también de la clase MensajeDos, dado que ambas clases tiene un método *setMensaje(...)* y en ambos casos el método tiene un parámetro de tipo String.
Si, en cambio, se indicara en el PC que interesan los métodos *setMensaje(int)*, no se produciría el entretejido con el advise "after" ya que no se produce la ***concordancia de patrones*** necesaria (es decir, considerar un mensaje que corresponda a cualquier clase, que se llame *setMensaje* y que tenga un parámetro de tipo String)
Ahora la salida será:

```
JAJAJA
JAJAJA
Hi!!!!
hola mundo
lunes   de primavera
***viento****y sol
***viento****y sol
```

```
***viento****y sol
```

Pruebe que sucede si cambia la declaración de PC por lo siguiente:

```
- pointcut saludar() :
            execution (void *.setMensaje(..))
- pointcut saludar() :
            execution (void *.setMensaje(..)) || call (void HolaMundo.mensaje());
- pointcut saludar() :
            execution (void *.setMensaje(..)) || call (void HolaMundo.mensaje(..));
```

¿Cual es el efecto del uso de (..)?

Ahora agregaremos otro PC y un advise *after:*

```
pointcut obtener():
            call (String *.*Mensaje());

after(): obtener(){
            System.out.println(" obtengo el mensaje de MensajeUno");
}
```

¿Que se obtiene?

El problema con el advise anterior es que siempre escribe "... MensajeUno". Tendría mas sentido si pidieramod lograr que el mensaje fuera " … MensajeUno" o " …. MensajeDos" dependiendo de cual sea el objeto *target,* o sea el objeto sobre el que se esta aplicando el advise.
Para conseguirlo es necesario trabajar con la ***información del contexto.***

```
public class HolaMundo {
      public static void main (String[] args){


            MensajeUno   menUno = new MensajeUno("lunes");
            MensajeDos   menDos = new MensajeDos("viento", 3);

            menUno.setMensaje(" de primavera");
            menDos.setMensaje("y sol");
            mensaje();
            mensaje(menUno);
            mensaje(menDos);
      }

      public static void mensaje(){
            System.out.println ("hola mundo");
      }

      public static void mensaje(Mensaje mUno){
            System.out.println (mUno.getMensaje());
      }

}
```

```
public interface Mensaje {

	public String getMensaje();
	public void setMensaje(String mens);
	public String getDatos();
}


public class MensajeUno implements Mensaje{

	private String uno;

	public MensajeUno (String cad){
			uno = cad;
	}

	public String  getMensaje(){
		return uno;
	}

	public String  getDatos(){
		return "MENSAJE UNO";
	}

	public void setMensaje(String otro){
		uno = uno + "   " + otro;
	}




}



public class MensajeDos implements Mensaje {
	private String dos;
	private int    repeticiones;

	public MensajeDos (String cad, int rep){
			dos = cad;
			repeticiones = rep;
	}

	public String  getMensaje(){
		return dos;
	}

	public String  getDatos(){
		return "MENSAJE DOS";
	}
```

```
	public void setMensaje(String otro){
		String aux = "***" + dos + "****" + otro;
		dos = aux;
		for (int i=1; i< repeticiones; i++)
			dos = dos + "\n" + aux;

	}

}


public aspect Saludos {
	pointcut saludar() :
		execution (void *.setMensaje(..)) || call (void HolaMundo.mensaje());

	pointcut obtener(Mensaje mensaje):
		call (String *.*Mensaje()) && target(mensaje);

		after(): saludar(){
			System.out.println(" JAJAJA");
		}

		after(Mensaje mensaje): obtener( mensaje){
			System.out.println(" obtengo el mensaje de" + mensaje.getDatos());
		}

		before(): execution (void HolaMundo.mensaje()){
			System.out.println("Hi!!!! ");
		}
}
```

Ejercicio 3
Considere la siguiente clase
a)

```
public abstract class Account {
	private float _balance;
	private int _accountNumber;

	public Account(int accountNumber) {
		_accountNumber = accountNumber;
	}

	public void credit(float amount) {
		setBalance(getBalance() + amount);
```

```
	}

	public void debit(float amount) throws InsufficientBalanceException {
		float balance = getBalance();
		if (balance < amount) {
			throw new InsufficientBalanceException
					("Balance total no suficiente");
		} else {
			setBalance(balance - amount);
		}
	}

	public float getBalance() {
		return _balance;
	}

	public void setBalance(float balance) {
		_balance = balance;
	}
}

public class InsufficientBalanceException extends Exception {

	public InsufficientBalanceException(String message) {
		super(message);
	}

}


public class SavingAccount extends Account {
	private static int MAXIMOEXT = 5;
	private int cantidad;
	public SavingAccount(int accountNumber) {
		super(accountNumber);
		cantidad = 0;
	}

	public void debit(float amount) throws InsufficientBalanceException {
		super(amount);
		cantidad --;
	}

}



public class Test {

		public static void main(String[] args)
		throws InsufficientBalanceException {
		SavingAccount account = new SavingAccount(12456);
		account.credit(100);
```

```
            account.debit(50);
            }

}


public aspect MinimumBalanceRuleAspect {
      private float Account._minimumBalance;
      public float Account.getAvailableBalance() {
            return getBalance() - _minimumBalance;
      }


      after(Account account):
            execution(SavingsAccount.new(..)) && this(account) {
                  account._minimumBalance = 25;
            }

      before(Account account, float amount) throws InsufficientBalanceException :
            execution(* Account.debit()) && this(account) && args(amount) {
                  if (account.getAvailableBalance() < amount) {
                                    throw new InsufficientBalanceException(
                                                "Insufficient available balance");
                  }
            }
}
```

Cree un aspecto “AspectoBancario” y considere el siguiente pointcut:

- pointcut creditOperations() : call(void Account.credit(float));
- considere los distintos tipos de “advice”: after(), before(), after() returning (int)
- utilice las siguientes expresiones para la signatura:

    - public void Account.set*(*)
    - public void Account.*()
    - public * Account.*()
    - public * Account.*(..)
    - *Account.*(..)
    - !public * Account.*(..)
    - public static void Test.main(String[] args)
    - *Account+.*(..)
    - public void Account.debit(float) throws InsufficientBalanceException
    - public Account.new()
    - public Account.new(int)
    - public Account.new(..)
    - public Account+.new(..)
    - public *Account.new(..)
    - public Account.new(..) throws InvalidAccountNumberException
    - private float Account._balance

- * Account.*
- !public static * banking..*.*
- public !final *.*

b) Considere el siguiente advice

```
…
before (Account cuenta, float monto):
        call (void Account.credit(float))
        && target (cuenta)
        && args (cantidad) {
                System.out.println("depositando" + cantidad + "en la cuenta" + cuenta);
        }
```

Ejercicio 4
Trabaje sobre los ejemplos disponibles en PEDCO de sincronización: lock, cyclic barrier. Analice la posibilidad de modularizar los ejemplos presentados aplicando aspectos e impleméntelo