UNIVERSIDAD CATÓLICA ANDRÉS BELLO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

Programación por Chequeo

(Apéndices)

Trabajo Especial de Grado

PRESENTADO ANTE LA

Universidad Católica Andrés Bello

COMO PARTE DE LOS REQUISITOS PARA OPTAR AL TÍTULO DE

Ingeniero en Informática

Autor: Paz Rojas, Juan Luis

Tutor: Torrealba, William

Septiembre de 2007

Contenido en apéndices

Apéndice A

Programación por chequeo en C#

A continuación se presentan las nuevas construcciones de las programación por chequeo aplicadas al lenguaje de programación C#. Cada construcción es estudiada en detalle, indicando su definición, formas de declaración y de utilización, también se especifican las reglas y consideraciones especiales que aplican a cada una de ellas.

A.1 Aspectos

Son una unidad programática capaz de controlar las entradas y salidas de una unidad funcional y que van enmarcadas dentro de una construcción orientada a objetos.

A.1.1 Descripción

En una unidad funcional los aspectos pueden realizar acciones iniciales y posteriores (en caso se ejecución normal o en caso de excepción) a la ejecución del

mismo, estas acciones se encuentran separadas del código y agrupadas en una construcción orientada a objetos, permitiendo al aspecto ser reutilizable en diferentes unidades funcionales.

Un aspecto se puede aplicar sobre una unidad funcional, ya sean métodos, propiedades o bloques de códigos.

A.1.2 Declaración

Se debe crear un contenedor para el aspecto, que podría ser una clase o una estructura,. El contenedor puede implementar más interfaces y si es una clase podría tener también una clase base de la que hereda. Se sugiere el uso del sufijo `Aspect` para nombrar los contenedores de aspectos.

Ejemplo:

```
class MiAspecto : Aspect {/*...*/}
static class MiAspecto : Aspect {/*...*/}
struct MiAspecto : Aspect {/*...*/}
```

Para dar soporte a los aspectos en el lenguaje se ha creado un nuevo tipo de miembro, de posible empleo en clases, estructuras e interfaces; así como los campos y los métodos son tipos de miembros utilizables en una clase, estructura o interfaz, los aspectos también lo son y se pueden declarar en los mismos lugares en donde es posible declarar un método.

Sintaxis general:

```
aspect(parámetros de ingreso):(parámetros de egreso)
{
  pre {/*...*/}
  post {/*...*/}
  handler {/*...*/}
```

```
}
```

La declaración del aspecto se hace empleando la palabra `aspect` y dentro de paréntesis se listan los parámetros de ingreso del aspecto, seguido de dos puntos y nuevamente dentro de paréntesis se listan los parámetros de egreso del aspecto; la lista de parámetros (ya sean de ingreso o de egreso) siguen la estructura y reglas que rigen a los métodos para declarar los parámetros que recibe, por lo que es válido el empleo de los modificadores `ref`, `out` y `params`.

Se puede hacer sobrecarga de aspectos tal como ocurre en los métodos, ya que los parámetros (de ingreso y de egreso) forman parte de la firma del mismo.

A.1.2.1 Los descriptores

El descriptor `pre` indica las operaciones que se deben realizar antes de la ejecución de la primera instrucción del código sobre el cual se aplica el aspecto.

El descriptor `post` indica las operaciones que se deben realizar después de la ejecución de la última instrucción del código sobre el cual se aplica el aspecto.

El descriptor `handler` indica las operaciones que se deben realizar cuando una excepción que no sea manejada dentro del código sobre el cual se aplica el aspecto atraviesa su ámbito, en el `handler` se debe hacer la captura de la excepción tal como se hace para capturarlas después de haber colocado un `try`, utilizando un `catch`, ya que la excepción está sin capturar.

Ejemplo:

```
handler {
  catch (Exception e) {
    // ...
    throw;
```

```
    }
}
```

En el `handler` no se puede detener la propagación de las excepciones, por lo que es necesario que se relance la excepción (preferiblemente con la sentencia `throw;` ya que esta preserva la pila de llamadas de la excepción) o se lance otra en su lugar.

Nota: En un aspecto no es necesario mencionar todos los descriptores.

A.1.2.2 Parámetros de ingreso

Son parámetros que existen al momento de entrar a la unidad funcional sobre la cual se aplica el aspecto, es decir, antes de la ejecución de la primera instrucción del código sobre el cual se aplica el aspecto deben poseer un valor; los parámetros de ingreso pueden ser de paso por valor (por defecto) o de referencia (empleando el modificador `out` ó `ref`) y pueden poseer una lista de argumentos variable (empleando el modificador `params`), como si de un método se tratase.

Los parámetros de ingreso están disponibles para su utilización en cualquiera de los descriptores.

A.1.2.3 Significado de los modificadores ref y out en parámetros de ingreso

Si el modificador del parámetro es:

- **Ninguno (no posee)** significa que al momento de ingresar a la unidad funcional sobre la cual se aplica el aspecto, el argumento ya posee un valor que el aspecto va a inspeccionar. El argumento se pasa por valor, por lo que cualquier cambio en este no es visible fuera del descriptor.

- **ref** significa que al momento de ingresar a la unidad funcional sobre la cual se aplica el aspecto, el argumento ya posee un valor que el aspecto va a inspeccionar y que puede cambiar (ya que el argumento se pasa por referencia).
- **out** significa que al momento de ingresar a la unidad funcional sobre la cual se aplica el aspecto, el argumento no posee un valor pero debe tener uno antes de ejecutarse la primera instrucción del código sobre el cual se aplica el aspecto, por lo que el aspecto debe darle un valor en el descriptor `pre`. En los descriptores `post` y `handler` este parámetro se comporta como si tuviera el modificador `ref`. Dicho de otra manera, los parámetros `out` deben ser inicializados por el aspecto en `pre` y conservan su modificabilidad (pero no es requerida su inicialización) en `post` y en `handler`.

A.1.2.4 Parámetros de egreso

Son parámetros que existen al momento de salir de la unidad funcional sobre la cual se aplica el aspecto, es decir, al momento de egresar de la unidad funcional deben poseer un valor; los parámetros de egreso pueden ser de paso por valor (por defecto) o de referencia (empleando el modificador `out` ó `ref`) y pueden poseer una lista de argumentos variable (empleando el modificador `params`), como si de un método se tratase.

Los parámetros de egreso están disponibles para su utilización en el descriptor `post`, pero no se pueden utilizar en los descriptores `pre` y `handler`.

A.1.2.5 Significado de los modificadores ref y out en parámetros de egreso

Si el modificador del parámetro es:

- **Ninguno (no posee)** significa que al momento de salir de la unidad funcional sobre la cual se aplica el aspecto, el argumento ya posee un valor que el aspecto va a inspeccionar. El argumento se pasa por valor, por lo que cualquier cambio en este no es visible fuera del descriptor.
- **ref** significa que al momento de salir de la unidad funcional sobre la cual se aplica el aspecto, el argumento ya posee un valor que el aspecto va a inspeccionar y que puede cambiar (ya que el argumento se pasa por referencia).
- **out** significa que al momento de salir de la unidad funcional sobre la cual se aplica el aspecto, el argumento no posee un valor pero debe tener uno antes de ejecutarse la primera instrucción después del código sobre el que se aplica el aspecto, por lo que el aspecto debe darle un valor en el descriptor `post`.

A.1.2.6 Modificadores

Un aspecto puede recibir los mismos modificadores que un método excepto el modificador `extern`, es decir, los modificadores válidos son: `public`, `protected`, `internal`, `private`, `static`, `virtual`, `sealed`, `new`, `override` y `abstract`, y su utilización se rige bajo las mismas reglas que deben seguirse en el caso de los métodos.

A.1.2.7 Declaración sin implementación

En el caso de no dar implementación a los descriptores se coloca punto y coma

después de enunciar todos los parámetros.

Sintaxis general:

```
aspect(parámetros de ingreso):(parámetros de egreso);
```

Esta forma se utiliza para declarar aspectos en interfaces o aspectos que lleven el modificador `abstract`.

A.1.2.8 Implementación explícita

Para realizar una implementación explícita de un aspecto, se omiten los modificadores de este y se antepone a la palabra `aspect` el nombre de la interfaz seguido de un punto.

Ejemplo:

```
IMiInterfaz.aspect():() {/* ... */}
```

A.1.2.9 Asignaciones preliminares

En un aspecto es posible realizar asignaciones preliminares a la ejecución del descriptor `pre`, estas asignaciones se realizan en el mismo nivel en el que se coloca la implementación de los descriptores, es decir, se coloca dentro de las llaves de implementación del aspecto, pero fuera de algún descriptor.

Nota: Estas asignaciones tienen el mismo efecto que si se hubiera hecho en la primeras líneas del descriptor `pre`.

Ejemplo:

```
struct DepositoAspect
{
```

```
    decimal balance_anterior;

    aspect (decimal balance, decimal importe):()
    {
      balance_anterior = balance;

      pre {
        if (importe < 0)
          throw new ArgumentOutOfRangeException(
            "importe", "importe no puede ser negativo"
          );
      }

      post {
        if (balance != balance_anterior + importe)
          throw new Exception(
            "El resultado del método es errado");
      }
    } // fin aspect
}
```

A.1.3 Utilización

Los aspectos se pueden utilizar en cualquier unidad funcional, tales como métodos, propiedades y bloques de códigos, su utilización se anuncia con la palabra `aspect` y listando cada uno de los aspectos por el nombre de la clase o estructura contenedora y dentro de los correspondientes paréntesis la lista de argumentos que requiere (opcionalmente se puede escribir un punto para separar a ambos), cada aspecto a aplicar se separa con coma, o se enuncia nuevamente con la palabra `aspect`.

A.1.3.1 Aspectos sobre métodos

Los aspectos a ser aplicados sobre un método (o constructor) se colocan después de cerrar el paréntesis de lista de parámetros y antes de abrir la llave de implementación de código; su utilización se anuncia usando la palabra `aspect` y luego se indica el nombre de la clase contenedora del aspecto estáticos,

opcionalmente se escribe un punto, y dentro de paréntesis la lista de argumentos que recibe.

En la lista de parámetros del aspecto a usar se puede utilizar los parámetros del método, literales y cualquier otro valor externo al método que sea accesible desde este.

Ejemplo:

```
void MiMetodo(int a, int b, int c, out int d)
  aspect MiAspecto.(a, b):(d), OtroAspecto(b, c):(d)
{/*...*/}


void OtroMetodo(ref string s)
  aspect ControlStringAspect(ref s):()
  aspect AlgunOtroAspecto.():()
{/*...*/}
```

A.1.3.2 Aspectos sobre descriptores de propiedades

La aplicación de los aspectos sobre las propiedades es similar al de los métodos, pero en las propiedades se utilizan sobre los descriptores de acceso y se coloca antes de la apertura de la llave que indica la implementación del descriptor; su utilización se anuncia empleando la palabra `aspect` y luego se indica el nombre de la clase contenedora del aspecto y dentro de paréntesis la lista de argumentos que recibe.

En la lista de parámetros del aspecto a usar se puede utilizar para el descriptor `get` el parámetro implícito `value`.

Ejemplo:

```
int Propiedad
{
```

```
  get aspect MiAspecto():()
  {/*...*/}

  set aspect MiAspecto(value):()
  {/*...*/}
}
```

A.1.3.3 Aspectos sobre descriptores de indizadores

Al igual que en los descriptores de propiedades es posible aplicar aspectos sobre los descriptores de indizadores.

Ejemplo en indizadores:

```
string this [int posicion]
{
  get aspect MiAspecto():()
  {/*...*/}

  set aspect MiAspecto(value):()
  {/*...*/}
}
```

A.1.3.4 Aspectos sobre redefinición de operadores

Al igual que en los métodos es posible aplicar aspectos sobre la redefinición de operadores (ya sean operadores unarios, binarios y de conversión).

Ejemplo:

```
public static Elemento operator ++ (Elemento operando)
  aspect MiAspecto(operando):()
{/*...*/}

public static Elemento operator + (Elemento op1, Elemento op2)
  aspect MiAspecto(op1, op2):()
{/*...*/}

public static implicit operator Tipo1(Elemento operando)
  aspect MiAspecto(operando):()
{/*...*/}

public static explicit operator Tipo2(Elemento operando)
  aspect MiAspecto(operando):()
```

```
{/*...*/}
```

A.1.3.5 Aspectos compuestos

Son aspectos cuya funcionalidad es el resultado de la combinación de la funcionalidad propia y de la funcionalidad de los otros aspectos que sobre este se aplican.

Los parámetros que se pueden utilizar en los aspectos componentes son literales, cualquier valor externo al aspecto compuesto que sea accesible desde los descriptores de este, también es posible usar los parámetros del aspecto compuesto como argumentos de los aspectos componentes, pero con una restricción, los parámetros de egreso del aspecto compuesto sólo son utilizables como argumentos de egreso en los aspectos componentes, aunque los parámetros de ingreso son utilizables como argumentos de ingreso y de egreso.

Ejemplo:

```
aspect(int parametro):()
  aspect MiAspecto(parametro, "Algún valor"):()
{
  pre {/*...*/}
  post {/*...*/}
  handler {/*...*/}
}
```

A.1.3.6 Aspectos sobre bloques de códigos

La aplicación de aspectos sobre bloques de código se hace a través de una extensión de la instrucción `using`. Se usa colocando la palabra `using` y opcionalmente dentro de paréntesis los argumentos de la instrucción, seguidamente se anuncia el uso del aspecto con la palabra `aspect` y se continúa de

la misma forma que se hace en un método.

Los argumentos de la instrucción `using` (de tener) se pueden utilizar como argumentos de ingreso y egreso en los aspectos, ya las instrucciones dadas como argumento a la instrucción `using` se ejecutan primero que la llamada al descriptor `pre` de los aspectos y la llamada al método `Dispose` ocurre después de haber culminado la ejecución de los descriptores `post` ó `handler` (según corresponda) de los aspectos.

Ejemplo:

```
{
  int x = 0, y;

  using
    aspect UnAspecto(x):(y)
  {
    otrasOpertaciones();
    // ...
    y = 3;
  }

  // ...

  using (MiObjeto z = new MiObjeto)
    aspect OtroAspecto(x,y):(z)
  { /* ... */ }
}
```

A.1.4 Orden de aplicación

Cuando se aplican varios aspectos sobre una misma unidad funcional, el orden de ejecución del descriptor `pre` es el mismo que el orden en que se enunciaron los aspectos, pero para los descriptores `post` y `handler` el orden es inverso; por los que al aplicar varios aspectos es conveniente ordenarlos del más general al más específico.

Ejemplo:

```
void MiMetodo()
   aspect Aspecto1():()
   aspect Aspecto2():()
   aspect Aspecto3():()
{
   // ... Contenido del método ...
}
```

Al invocar a `MiMetodo` se ejecutará primero el descriptor `pre` de `Aspecto1`, luego el de `Aspecto2`, y por último el de `Aspecto3`; luego se ejecuta el contenido del método, de no haber sido lanzada una excepción se ejecuta el descriptor `post` de `Aspecto3` en caso contrario se ejecuta el descriptor `handler` de este, luego de no haber sido lanzada una excepción se ejecuta el descriptor `post` de `Aspecto2` en caso contrario se ejecuta el descriptor `handler` de este y por último de no haber sido lanzada una excepción se ejecuta el descriptor `post` de `Aspecto1` en caso contrario se ejecuta el descriptor `handler` de este.

El resultado es similar de haber escrito:

```
void MiMetodo() {
   using aspect Aspecto1():() {
      using aspect Aspecto2():() {
         using aspect Aspect3():() {
            // ... Contenido del método ...
         }
      }
   }
}
```

A.1.5 Aspectos instanciados

Es posible utilizar aspectos para los que debe existir una instancia previa, para ello, después haber anunciado la aplicación de aspectos con la palabra `aspect`, en vez de escribir el nombre de la clase contenedora del aspecto estático, se usa

algunas de las siguientes construcciones:

- Para aplicar un aspecto instanciado previamente creado, se coloca el nombre de la variable que referencia a la instancia del contenedor del aspecto y luego los argumentos que recibe el aspecto (opcionalmente se puede escribir un punto para separar a ambos).

  Ejemplo:

```
void HacerAlgo(string s, AspectoNoEstatico a)
  aspect a(s):()
{/*...*/}


void HacerAlgo2(string s, OtroObjeto obj)
  aspect obj.AspectoUsado(s):()
{/*...*/}


void HacerAlgo3(string s, OtroObjeto obj)
  aspect obj.GetAspectoUsado()(s):()
{/*...*/}
```

  Importante: La instancia del aspecto se va a recuperar cada vez que se invoque a un descriptor.

  Nota: Estas son construcciones válidas para ser utilizadas en la composición de aspectos.

- Para crear una instancia de un aspecto, se puede declara la variable que referenciará al contenedor del aspecto o (de ya estar declarada) simplemente escribir el nombre de la variable, luego se coloca los argumentos del aspecto (opcionalmente se puede escribir un punto para separar a ambos), posteriormente se escribe el signo = (igual) y se hace la inicialización de objeto contenedor del aspecto, bien sea llamando a

un constructor o asignado un objeto preexistente.

Ejemplo:

```
void HacerAlgo1(string s)
  aspect AspectoNoEstatico a(s):() = new AspectoNoEstatico()
{/*...*/}


void HacerAlgo2(string s, AspectoNoEstatico a)
  aspect AspectoNoEstatico b(s):() = a
{/*...*/}


void HacerAlgo3(string s, OtroObjeto obj)
  aspect AspectoNoEstatico b(s):() = obj.GetAspectoUsado()
{/*...*/}


void HacerAlgo4(string s, OtroObjeto obj)
  aspect AspectoNoEstatico b(s):() = obj.AspectoUsado
{/*...*/}
```

Importante: La instancia del contenedor del aspecto se va a recuperar una única vez (antes de llamar al descriptor `pre`) y se le asigna a la variable declarada, la invocación de cada descriptor se hace utilizando el valor de la variable declarada en la aplicación.

Nota: Estas construcciones no se pueden utilizar en la composición de aspectos.

- Se puede crear una instancia anónima del contenedor de un aspecto (creada in situ), para ello se crea la instancia de la clase contenedora del aspecto, luego se coloca los argumentos del aspecto (opcionalmente se puede escribir un punto para separar a ambos).

  Ejemplo:

  ```
  void HacerAlgo(string s, OtroObjeto obj)
    aspect new AspectoNoEstatico()(s):()
  ```

```
{/*...*/}
```

Importante: La instancia del contenedor del aspecto se va a crear una única vez (antes de llamar al descriptor `pre`) y se le asigna a la variable temporal, la invocación de cada descriptor se hace utilizando el valor de esta variable temporal.

Nota: Estas construcciones no se pueden utilizar en la composición de aspectos.

Estas reglas se aplican a todos los lugares donde es utilizable un aspecto, excepto en la composición de aspectos donde aplican algunas restricciones.

Ejemplo:

```
{
  string s = "conexión";

  using
    aspect Transaccion t():() = new Transaccion(s)
  {
    otrasOpertaciones();
    t.Commit();

    // ...
  }
}


void MiMetodo(AspectoNoEstatico a)
  aspect a():(), new OtroAspectoNoEstatico()():()
{/*...*/}
```

A.1.5.1 Obtención de instancias

El código utilizado para obtener la instancia del aspecto a ser utilizado se ejecuta cada vez que se invoque a un descriptor, para cambiar este comportamiento se debe emplear las construcciones que permiten declarar una variable que

referenciará al contenedor del aspecto o crear una instancia anónima, en estos casos la instancia se obtendrá antes de la llamada al descriptor `pre`.

A.1.6 El valor de retorno

El valor que retorna un método o una propiedad `get` se puede obtener empleando la palabra `returned`. Esto solo aplica en los aspectos aplicados directamente sobre métodos y propiedades `get` (aplicados antes de la apertura de la apertura de la llave de implementación), no sobre bloques de códigos en general.

Ejemplo:

```
int UnMetodo(string s)
  aspect MiAspecto(s):(ref returned)
{/*...*/}


int OtroMetodo(string s, AspectoNoEstatico a)
  aspect a(s):(ref returned)
{/*...*/}


int Propiedad
{
  get aspect MiAspecto( ):(returned) {/*...*/}
  set aspect MiAspecto(value):( ) {/*...*/}
}
```

Si el último aspecto (en orden de anunciación) que utiliza `returned` como parámetro de egreso es empleado con el modificador `out`, entonces, en la implementación del método no es necesario retornar un valor, incluso se podría usar la sentencia `return;` sin valor de retorno; el valor que retorna el método o propiedad `get` se obtiene del aspecto que tiene en sus argumentos de egreso a `out returned`.

Ejemplo:

```
int MiMetodo(string s)
   aspect MiAspecto(s):(out returned)
{
   // ...
   return;
}
```

A.1.7 Utilización de la sentencia de retorno back

La sentencia `back;` provoca la terminación normal del descriptor del aspecto, tal como si la ejecución alcanzara el final del código contenido en el descriptor. Solo es utilizable esta sentencia en los descriptores `pre` y `post`, en `handler` no es posible su uso ya que es requerido que se culmine con el lanzamiento de una excepción.

Sintaxis general:

```
back;
```

Ejemplo:

```
aspect(int[] arreglo):()
{

   pre {

      for(int i = 0; i < arreglo.Lenght; i++)

         if(arreglo[i] > 7)
            back;// Hace que se termine pre y no se continúe con
                 // la comprobación, pasando entonces a ejecutarse
                 // la primera instrucción del bloque de código
                 // sobre el que se aplica el aspecto

         else {
            // ...
         }
   }

   post {/*...*/}

   handler {/*...*/}
}
```

Nota: No se permite la utilización de la sentencia de retorno `return` en ninguno de los descriptores de un aspecto, `back;` es la única sentencia de retorno válida.

A.1.8 Aspectos compuestos

Son aspectos que se componen de otros aspectos, tal que la funcionalidad de este sea la combinación de la funcionalidad de los otros aspectos que lo componen y la funcionalidad propia.

Ejemplo:

Si se tiene tres aspectos. `A`, `B` y `C`, y el aspecto `C` se compone con los aspectos `A` y `B`:

```
static class A {

   public static aspect():() {

      pre { Algo.HacerPreA(); }
      post { Algo.HacerPostA(); }

      handler {
         catch(AException) { Algo.HacerHandlerA(); }
      }
   }
}


static class B {

   public static aspect():() {

      pre { Algo.HacerPreB(); }
      post { Algo.HacerPostB(); }

      handler {
         catch(BException) { Algo.HacerHandlerB(); }
      }
   }
}
```

```
static class C {

  public static aspect():()
    aspect A():()
    aspect B():()
  {


    pre { Algo.HacerPreC(); }
    post { Algo.HacerPostC(); }

    handler {
      catch(CException) { Algo.HacerHandlerC(); }
    }
  }
}
```

El comportamiento del aspecto `C` es que para cada uno de los descriptores se realiza la acción cada descriptor equivalente según el orden de aparición para el descriptor `pre` y en orden inverso para `post` y `handler`, por lo que un equivalente funcional a `C` sería:

```
static class C {

  public static aspect():() {

    pre {
      Algo.HacerPreA();
      Algo.HacerPreB();
      Algo.HacerPreC();
    }

    post {
      Algo.HacerPostC();
      Algo.HacerPostB();
      Algo.HacerPostA();
    }

    handler {
      catch(CException) { Algo.HacerHandlerC(); }
      catch(BException) { Algo.HacerHandlerB(); }
      catch(AException) { Algo.HacerHandlerA(); }
    }
  }
}
```

A.1.8.1 Orden de composición

El orden de composición de aspectos es similar al orden de aplicación de aspectos; se aplican primero los descriptores `pre` en el orden en que fueron enunciados los aspectos componentes y por último el descriptor `pre` del aspecto compuesto, para los descriptores `post` y `handler` el orden es el inverso.

Ejemplo:

```
void MiMetodo()
  aspect Aspecto1():()
  aspect Aspecto2():()
  aspect Aspecto3():()
{
  // ... Contenido del método ...
}
```

Es similar a haber declarado el aspecto:

```
static class MiAspecto
{

  public aspect():()
    aspect Aspecto1():()
    aspect Aspecto2():()
    aspect Aspecto3():()
  { }
}
```

Y en `MiMetodo` haberlo usado:

```
void MiMetodo()
  aspect MiAspecto():()
{
  // ... Contenido del método ...
}
```

El resultado es similar de haber escrito:

```
void MiMetodo() {
  using aspect Aspecto1():() {
    using aspect Aspecto2():() {
```

```
        using aspect Aspect3():() {
          // ... Contenido del método ...
        }
      }
    }
}
```

A.1.8.2 Alterar el orden de composición

Se puede alterar la posición que ocupará el código del aspecto compuesto frente a los aspectos componentes utilizando la palabra `here` como nombre de un aspecto y sin colocar los paréntesis de argumentos en la lista de aspectos que lo componen.

De forma implícita se asume que `here` es el último de la lista al menos que sea usado de forma explícita, por lo que colocar `here` al final de la lista o no colocarlo es exactamente equivalente.

Ejemplo:

Si se tiene tres aspectos. `A`, `B` y `C`, y el aspecto `C` se compone con los aspectos `A` y `B`, pero el orden de composición deseado es primero al aspecto `A` luego el contenido propio del aspecto `C` y por último el aspecto `B`:

```
static class A {

  public static aspect():() {

    pre { Algo.HacerPreA(); }
    post { Algo.HacerPostA(); }

    handler {
      catch(AException) { Algo.HacerHandlerA(); }
    }
  }
}

static class B {
```

```
  public static aspect():()  {

    pre { Algo.HacerPreB(); }
    post { Algo.HacerPostB(); }

    handler {
      catch(BException) { Algo.HacerHandlerB(); }
    }
  }
}


static class C {

  public static aspect():()
    aspect A():()
    aspect here
    aspect B():()
  {

    pre { Algo.HacerPreC(); }
    post { Algo.HacerPostC(); }

    handler {
      catch(CException) { Algo.HacerHandlerC(); }
    }
  }
}
```

El comportamiento del aspecto `C` es que para cada uno de los descriptores se realiza la acción cada descriptor equivalente según el orden de aparición para el descriptor `pre` y en orden inverso para `post` y `handler`, pero hay que notar que `here` indica la posición del contenido de aspecto `C` y deja de estar implícitamente al final; por lo que un equivalente funcional a `C` sería:

```
static class C
{

  public static aspect():()
  {

    pre {
      Algo.HacerPreA();
      Algo.HacerPreC();
      Algo.HacerPreB();
    }
```

```
      post {
        Algo.HacerPostB();
        Algo.HacerPostC();
        Algo.HacerPostA();
      }

      handler {
        catch(BException) { Algo.HacerHandlerB(); }
        catch(CException) { Algo.HacerHandlerC(); }
        catch(AException) { Algo.HacerHandlerA(); }
      }
    }
}
```

A.1.8.3 Composición recursiva

Se debe tener cuidado al crear aspectos que se comporten de forma recursiva, en aspectos compuestos, ya sea recursividad directa o indirecta ya que se puede caer en ciclos infinitos.

A.1.9 Sobrecarga de aspectos

Los aspectos son sobrecargables en función de la lista de su lista de parámetros de ingreso y egreso; tal como ocurre en los métodos.

Ejemplo:

```
static class MiAspecto
{

  public static aspect():()
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
  }

  public static aspect(string s):()
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
```

```
    }

    public static aspect():(string s)
    {
      pre {/*...*/}
      post {/*...*/}
      handler {/*...*/}
    }
}


class MiClase
{

    void MiMetodo1()
      aspect MiAspecto():()
    {/*...*/}

    void MiMetodo2()
      aspect MiAspecto("parámetro"):()
    {/*...*/}

    void MIMetodo3()
      aspect MiAspecto():("parámetro")
    {/*...*/}
}
```

A.1.9.1 Intercomposición de aspectos sobrecargados

Para la composición de un aspecto con una de sus sobrecargas, en vez de dar el nombre de la case contenedora o de la variable que referencia a la clase contenedora del aspecto, simplemente se colocan los argumentos del aspecto; y opcionalmente se puede anteponer a los argumentos:

- el nombre de la clase actual, para utilizar un aspecto estático.
- las palabras `this` ó `base`, para utilizar aspectos instanciados; conservando estas palabras el significado tradicional que siempre han tenido.

Ejemplo:

```
class MiAspecto
```

```
{

  aspect():()
    // Se compone con el aspecto no estático:
    // aspect(string a):()
    aspect ("Por defecto"):()
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
  }

  aspect(string a):()
    // Se compone con el aspecto no estático:
    // aspect(string b, string c):()
    aspect this(a, "Por defecto"):()
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
  }

  aspect(string b, string c):()
    // Se compone con el aspecto estático:
    // static aspect(string d, string e):(string f)
    aspect (b, c):("Por defecto")
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
  }

  static aspect(string d, string e):(string f)
    // Se compone con el aspecto estático:
    // static aspect(string g, string h):(string i, string j)
    aspect MiAspecto(d, e):(f, "Por defecto")
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
  }

  static aspect(string g, string h):(string i, string j)
    // Se compone con el aspecto estático:
    // static aspect(string k, string l)
    //   :(string m, string n, string o)
    aspect (g, h):(i, j, "Por defecto")
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
  }
```

```
    static aspect(string k,string l):(string m,string n,string o)
    {
      pre {/*...*/}
      post {/*...*/}
      handler {/*...*/}
    }
}
```

A.2 Aspectos de inspección

Son un tipo de aspectos que no pueden provocar la finalización del bloque de código, o el retorno del método o propiedad sobre el cual se aplica.

A.2.1 Descripción

Los aspectos de inspección cumplen las siguientes reglas:

- El código sobre el que se aplica el aspecto siempre se ejecuta al menos que en el descriptor `pre` se lance una excepción.
- El descriptor `handler` siempre lanza una excepción, ya sea que relance la excepción que dio el motivo de su ejecución, o que inicie otra excepción.

Los aspectos de inspección corresponden a los aspectos descritos hasta ahora.

A.2.2 Declaración

Hay dos formas equivalentes de declarar aspectos de inspección:

- Implícitamente, en donde no se indica el tipo de aspecto (es la forma usada hasta ahora)

  ```
  aspect(parámetros de ingreso):(parámetros de egreso)
  {
  ```

```
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
}
```

- Explícitamente, en donde el tipo de aspecto es indicado en la declaración de este; esto se consigue agregando dos puntos después del cierre del paréntesis de parámetros de egreso y seguidamente colocando la palabra `noforce`

```
aspect(parámetros de ingreso):(parámetros de egreso):noforce
{
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
}
```

Indistintamente de cuál de las dos formas se haya usado el resultado es equivalente, ya que la primera es una forma abreviada de la segunda.

Ejemplo:

```
static class MiAspecto
{

  public aspect ():()
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
  }
}
```

Es equivalente a:

```
static class MiAspecto
{

  public aspect ():():noforce
  {
    pre {/*...*/}
    post {/*...*/}
    handler {/*...*/}
```

```
  }
}
```

A.2.3 Utilización

Hay dos formas equivalentes de utilizar aspectos de inspección:

- Implícitamente, en donde no se indica el tipo de aspecto (es la forma utilizada hasta ahora)

  `(`*`parámetros de ingreso`*`):(`*`parámetros de egreso`*`)`

- Explícitamente, en donde el tipo de aspecto es indicado en la declaración de este; esto se consigue agregando dos puntos después del cierre del paréntesis de parámetros de egreso y seguidamente colocando la palabra `noforce`

  `(`*`parámetros de ingreso`*`):(`*`parámetros de egreso`*`):`**`noforce`**

Indistintamente de cuál de las dos formas se haya usado el resultado es equivalente, ya que la primera es una forma abreviada de la segunda.

Ejemplo:

```
using
  aspect MiAspecto():()
{
  // ...
}
```

Es equivalente a:

```
using
  aspect MiAspecto():():noforce
{
  // ...
}
```

A.3 Aspectos retornantes

Son aspectos que tienen la capacidad de forzar culminación normal del bloque de código sobre el que se aplica, en un método o propiedad fuerza el retorno.

A.3.1 Descripción

Los aspectos retornantes pueden forzar el retorno con uno o varios valores; como pueden forzar el retorno sin retornar algún valor, solamente provocando la culminación normal del bloque de código.

Los aspectos retornantes difieren de los aspectos de inspección en:

- El descriptor `pre` puede forzar la culminación normal de la unidad funcional, lo que abre la posibilidad que el código sobre el que se aplica el aspecto no se ejecute, sin que el descriptor `pre` se lance una excepción.
- El descriptor `handler` puede contener la excepción que dio inicio a su ejecución y forzar a la culminación normal de la unidad funcional, por lo que el descriptor `handler` no siempre lanza una excepción.

A.3.2 Declaración

Los aspectos retornantes siempre se deben declarar de forma explícita; esto se consigue agregando dos puntos después del cierre del paréntesis de parámetros de egreso y seguidamente colocando la palabra `force`

Sintaxis general:

```
(parámetros de ingreso):(parámetros de egreso):force
```

Ejemplo:

```
aspect(parámetros de ingreso):(parámetros de egreso):force
{
  pre {/*...*/}
  post {/*...*/}
  handler {/*...*/}
}
```

Todos los parámetros de egreso deben ser de paso por referencia, por lo que deben poseer el modificador `ref` ó el modificador `out`, no se admiten parámetros de egreso sin modificador, o con el modificador `params`.

A.3.3 Utilización

Los aspectos retornantes siempre se deben utilizar de forma tal que expliciten su tipo; esto se consigue agregando dos puntos después del cierre del paréntesis de parámetros de egreso y seguidamente colocando la palabra `force`

Ejemplo:

```
using
  aspect MiAspecto():():force
{
  // ...
}
```

A.3.4 Forzar el retorno

Para forzar el retorno se usa la sentencia `return;` pero en ningún caso recibe un argumento.

Sintaxis general:

```
return;
```

Para poder llamar a la sentencia `return;` es necesario asignarle un valor a todos los parámetros de egreso que tengan el modificador `out`, en los descriptores `pre` y `handler` también es necesario asignarle un valor a los parámetros de egreso que tengan el modificador `ref` ya que no poseen alguno; es decir, en `pre` y `handler` hay que asignarle un valor a todos los parámetros de egreso antes de llamar a `return;` pero en `post` sólo es necesario en aquellos parámetros que posean el modificador `out`.

Nota: Cualquier valor asignado a los parámetros de egreso en los descriptores `pre` y `handler` no es visible fuera del aspecto si no es invocada la sentencia `return;`

Importante: Si se llama a `return;` en `pre` no se ejecutará ningún otro descriptor del aspecto.

Importante: El comportamiento de las instrucciones `return;` y `back;` en en descriptor `post` es idéntico, culminan la ejecución del descriptor y permite que continúe el retorno.

Ejemplo:

```
static class DetectarErrorAspect
{

  public aspect (string s):(out int retorno):force
  {

    pre {
      if (s == null) {
        retorno = -1; // Error: s es nulo
        return;

      } else if (s.Length <= 0) {
        retorno = -2; // Error: s está vacío
        return;
      }
    }
```

```
      post {
        retorno = 0;   // Resultado correcto
      }

      handler {
        catch {
          retorno = -3; // Error: otro tipo de error
        }
      }
    }
}


class MiClase
{

  static int MiMetodo(string s)
    aspect DetectarErrorAspect(s):(out returned):force
  {

    if (s == "malo")
      throw new Exception();
    else
      Console.WriteLine("   El texto es: {0}", s);
  }

  static void Main()
  {

    int resultado;

    Console.WriteLine("Llamando a MiMetodo con s = \"Hola\"");
    resultado = MiMetodo("Hola");
    Console.WriteLine("   El resuldado es: {0}\n", resultado);

    Console.WriteLine("Llamando a MiMetodo con s = null");
    resultado = MiMetodo(null);
    Console.WriteLine("   El resuldado es: {0}\n", resultado);

    Console.WriteLine("Llamando a MiMetodo con s = \"\"");
    resultado = MiMetodo("");
    Console.WriteLine("   El resuldado es: {0}\n", resultado);

    Console.WriteLine("Llamando a MiMetodo con s = \"malo\"");
    resultado = MiMetodo("malo");
    Console.WriteLine("   El resuldado es: {0}\n", resultado);
  }
}
```

El resultado de ejecutar el ejemplo es:

```
Llamando a MiMetodo con s = “Hola”
   El texto es: Hola
   El resuldado es: 0

Llamando a MiMetodo con s = null
   El resuldado es: -1

Llamando a MiMetodo con s = “”
   El resuldado es: -2

Llamando a MiMetodo con s = “malo”
   El resuldado es: -3
```

A.3.5 Sobrecarga de aspectos retornantes y de inspección

La distinción entre `force` y `noforce` provoca sobrecarga en los aspectos, por lo que es posible tener dos aspectos en el mismo contenedor y la misma lista de argumentos que solo se diferencien por su tipo: si es retornante o si es de inspección.

A.4 Aspectos anónimos

Son aspectos cuya implementación se realiza in situ, en el lugar donde se emplea. Los aspectos anónimos carecen de contenedor, tienen la capacidad de capturar las variables externas y pueden forzar el retorno o la culminación del bloque de código.

A.4.1 Descripción

Con aspectos anónimos es posible crear y utilizar aspectos retornantes implementados in situ, sin tener que crear un contenedor y una implementación separada a su utilización. Los aspectos anónimos son aspectos retornantes, que adicionalmente tienen pleno acceso a las variables externas a los descriptores de los

aspectos.

A.4.2 Declaración

Un aspecto anónimo se puede emplear en los mismos lugares donde es aplicable un aspecto, pero, en vez de indicar el nombre de la clase contenedora y los argumentos que recibe este, se coloca dentro de llaves la implementación de los descriptores del aspecto.

Ejemplo: Utilizando aspectos anónimos, `MiMetodo` en el ejemplo de forzar el retorno en aspectos retornantes se pudiera haber escrito como

```
int MiMetodo(string s)
  aspect
  {

    pre {
      if (s == null)
        return -1;
        // Error: s es nulo
      else if (s.Length <= 0)
        return -2;
        // Error: s está vacío
    }

    post { return 0; // Resultado correcto }

    handler {
      catch {
        return -3;  // Error: otro tipo de error
      }
    }
  } // fin aspect

{
  if (s == “malo”)
    throw new Exception();

  else
    Console.WriteLine(“   El texto es: {0}”, s);
}
```

A.4.3 Utilización

Se puede utilizar aspectos anónimos en los mismos lugares en los que se puede aplicar un aspecto, y en su implementación se tiene acceso a todos los elementos que a ese nivel se tiene visibilidad; si se aplica sobre o dentro de un método o propiedad, se tiene acceso a los parámetros de este y adicionalmente se puede utilizar la sentencia `return` dentro de los descriptores del aspecto anónimo.

Ejemplo:

```
void Ejemplo()
{
  int x = 0, y;

  using
    aspect UnAspecto():()
    aspect
    {

      pre {
        if(x<0) throw new ArgumentOutOfRangeException(
             “x”, “x no puede ser negativo”
          );
      }

      post {
        if(y<0) throw new Exception(
             “Ha ocurrido un error y el resultado es errado”
          );
      }
    } // fin aspect

  {
    otrasOpertaciones();
    // ...
    y = 3;
  }

  // ...

  using (MiObjeto z = new MiObjeto())
    aspect
      OtroAspecto(x,y):(z),
      {
        pre {
```

```
            if(x==y)
              throw new Exception("'x' e 'y' son iguales");
          }
        },
        AspectoAdicional():()

  { /* ... */ }
}
```

A.4.4 Utilización de la sentencia de retorno return

La sentencia `return` se puede utilizar dentro de los descriptores de un aspecto anónimo si en el ámbito en el cual se aplica el aspecto es posible su empleo; es decir, si el aspecto anónimo se aplica sobre un método o propiedad, o se aplica dentro de algún bloque de código contenido por el método o propiedad.

La sentencia `return` conserva su significado tradicional, es decir provoca el retorno del método o del descriptor de propiedad; si el método o el descriptor de propiedad retorna algún valor entonces la sentencia `return` recibe un argumento en caso contrario no.

Ejemplo:

```
class MiClase {

  void MiMetodo1(int argumento)
    aspct {
      pre { if (argumento < 0) return; }
    }
  {/*...*/}

  int MiMetodo2(int argumento)
  {
    using
      aspect { pre { if (argumento < 0) return 0; } }
    {/*...*/}
  }
}
```

A.4.5 Variables internas

En los aspectos anónimos es posible declarar variables internas al aspecto, a las que el elemento sobre el cual se aplica no tiene visibilidad, pero que dichas variables mantienen su valor mientras sea posible aplicar un descriptor del aspecto (aparte de `pre`).

Nota: No tiene sentido declarar una variable interna si el aspecto anónimo no posee un descriptor `post` o `handler` en su implementación.

Las variables internas de aspectos anónimos extienden las capacidades de las asignaciones preliminares de los aspectos, permitiendo declarar variables. Las variables internas se declaran y opcionalmente se inicializan en el mismo nivel en el que se coloca la implementación de los descriptores, es decir, se coloca dentro de las llaves de implementación del aspecto, pero fuera de algún descriptor.

Nota: Declarar e inicializar en la misma sentencia tiene el mismo efecto que si se hubiera declarado la variable interna y en el descriptor `pre` se hubiera inicializado.

Ejemplo: Supóngase una clase que representa una cuenta bancaria que cuenta con un método `Depositar` que recibe un importe como argumento y agrega ese importe al saldo

```
bool Depositar(decimal importe)
  aspect
  {

     decimal balance_anterior = balance;

     pre {
       if (importe < 0)
          throw new ArgumentOutOfRangeException(
```

```
            "importe", "importe no puede ser negativo"
          );
      }

      post {
        if (balance != balance_anterior + importe)
          throw new Exception(
            "El resultado del método es errado"
          );
      }
    } // fin aspect

{
  AgregarDeposito(importe);
}
```

Otra forma de haber resuelto este problema es haber utilizado un aspecto instanciado de inspección contenido en una estructura con un campo que equivaldría a la variable interna. Esta segunda forma permite la reutilización del aspecto.

De esta otra forma el método `Depositar` queda como:

```
bool Depositar(decimal importe)
  aspect new DepositoAspect()(balance, importe):()
{
  AgregarDeposito(importe);
}
```

Y el aspecto utilizado es:

```
struct DepositoAspect
{

  decimal balance_anterior;

  aspect (decimal balance, decimal importe):()
  {
    balance_anterior = balance;

    pre {
      if (importe < 0)
        throw new ArgumentOutOfRangeException(
          "importe", "importe no puede ser negativo"
        );
```

```
        }

        post {
          if (balance != balance_anterior + importe)
            throw new Exception(
              "El resultado del método es errado");
        }
      } // fin aspect
    }
```

A.5 Aspectos desmembrados

Son aspectos cuya implementación del sus descriptores se hace in situ, en el lugar donde se emplea, no requieren de definir el aspecto como tal y mucho menos un contenedor. En estos descriptores se puede capturar las variables externas y forzar el retorno o culminación del bloque de código.

A.5.1 Descripción

Con los aspectos desmembrados es posible implementar in situ el descriptor de un aspecto, si tener que crear un aspecto y mucho menos un contenedor para este. Estos descriptores se comportan como si estuvieran dentro de un aspecto retornante, que adicionalmente tiene pleno acceso a las variables externas.

A.5.2 Declaración

Para emplear un aspecto desmembrado, en vez de anunciar la aplicación de un aspecto con la palabra `aspect`, se coloca el nombre del descriptor y dentro de llaves la implementación del mismo. Al igual que en la aplicación de aspectos, se pueden separar varias implementaciones del descriptor con coma; o se puede enunciar el descriptor nuevamente y realizar otra implementación que

complementa a la primera.

Ejemplo: Utilizando aspectos desmembrados, `MiMetodo` en el ejemplo de forzar el retorno en aspectos retornantes se pudiera haber escrito como

```
int MiMetodo(string s)

  pre {
    if (s == null)
      return -1;  // Error: s es nulo
    else if (s.Length <= 0)
      return -2;  // Error: s está vacío
  }

  post { return 0; // Resultado correcto }

  handler {
    catch {
      return -3;  // Error: otro tipo de error
    }
  }

{
  if (s == “malo”)
    throw new Exception();
  else
    Console.WriteLine(“   El texto es: {0}”, s);
}
```

A.5.3 Utilización

Se puede emplear aspectos desmembrados en los mismos lugares en los que se puede aplicar un aspecto, y en su implementación se tiene acceso a todos los elementos que a ese nivel se tiene visibilidad; si se aplica sobre o dentro de un método o propiedad, se tiene acceso a los parámetros de este y adicionalmente se puede utilizar la sentencia `return` dentro de los descriptores del aspecto desmembrado.

Ejemplo:

```
void Ejemplo()
{
  int x = 0, y;

  using
    aspect UnAspecto():()
    pre {
      if(x<0) throw new ArgumentOutOfRangeException(
           “x”, “x no puede ser negativo”
        );
    }

    post {
      if(y<0) throw new Exception(
           “Ha ocurrido un error y el resultado es errado”
        );
    }

  {
    otrasOpertaciones();
    // ...
    y = 3;
  }

  // ...

  using (MiObjeto z = new MiObjeto)
    aspect OtroAspecto(x,y):(z)
    pre{if(x==y) throw new Exception(“‘x’ e ‘y’ son iguales”);}
    aspect AspectoAdicional():()

  {
    // ...
  }
}
```

A.5.4 Orden de aplicación

El orden de aplicación sigue las mismas reglas de orden de aplicación de aspectos, el orden de ejecución del descriptor `pre` es el mismo que el orden en que se enunciaron (bien sea dentro de un aspecto, o porque se enunció el descriptor), pero para los descriptores `post` y `handler` el orden es inverso; por los que es conveniente ordenarlos del más general al más específico.

Ejemplo:

```
void MiMetodo()
   aspect Aspecto1():()
   pre { /* pre1 */ }
   post { /* post1 */ }
   handler { /* handler1 */ }
   aspect {pre{/*pre2*/} post{/*post2*/} handler{/*handler2*/}},
     Aspecto2():()

{
   // ... Contenido del método ...
}
```

El resultado es similar de haber escrito:

```
void MiMetodo()
{

   using aspect Aspecto1():() {
     using pre { /* pre1 */ } {
       using post { /* post1 */ } {
         using handler { /* handler1 */ } {
           using aspect {
             pre{/*pre2*/}
             post{/*post2*/}
             handler{/*handler2*/} }
           {
             using Aspecto2():() {
               // ... Contenido del método ...
             }
           }
         } // fin de handler1
       } // fin de post1
     } // fin de pre1
   } // fin de Aspecto1():()
}
```

A.6 Instrucciones de chequeo

Son instrucciones que permiten realizar una serie de pruebas de forma secuencial dentro de una misma construcción.

A.6.1 Descripción

Son instrucciones que permiten realizar una serie de pruebas en secuencia sin tener que escribir instrucciones `if` anidadas, en donde se establece que la siguiente prueba no se evalúa al menos que la anterior haya sido superada, y donde basta con que falle una prueba para que de por no superado todo el conjunto.

A.6.2 Instrucción checkis

La instrucción `checkis` realiza una serie de pruebas, y retorna un booleano que indica si fueron superadas o no, donde la primera prueba que falle provoca que la instrucción retorne `false`, de superarse todas las pruebas la instrucción retorna `true`. Cada prueba es una expresión que debe retornar un valor booleano que indica si fue superada o no.

Sintaxis general:

```
checkis( prueba )
checkis( prueba1, prueba2, ... )
```

Ejemplo:

```
bool UnMetodo(int numero)
{

   return checkis(
     numero > 0
   );
}
```

Al ejecutar `UnMetodo(5)` este retorna `true`, pero si se ejecuta `UnMetodo(-5)` este `retorna false`

Ejemplo:

```
bool OtroMetodo(int numero)
{

   return checkis(
      numero > 0,
      (numero % 2) == 0
   );
}
```

En este caso la instrucción `checkis` realiza dos pruebas en forma secuencial: primero comprueba que `numero > 0` y si aprueba entonces realiza la segunda prueba, si la primera falla la instrucción retorna `false` y la segunda prueba no llega a ser ejecutada.

A.6.3 Instrucción check

La instrucción `check` realiza una serie de pruebas, en donde cada prueba inicia una excepción en caso de que la prueba falle, por lo que la prueba siguiente es ejecutada solo si la prueba anterior es superada.

Sintaxis general:

```
check( prueba )
check( prueba1, prueba2, ... )
```

Cada prueba es una instrucción que debe iniciar una excepción para indicar que la prueba no ha sido superada, por lo que es posible llamar a un método en una prueba. También es posible escribir una prueba que en funsión de un resultado booleano inicie una excepción, para ello se escribe la expresión que retorne un booleano, seguidamente se escribe `else` para indicar la acción a ejecutarse en caso de falla, es decir, que la expresión retorne `false`, esta acción se debe iniciar una excepción.

Sintaxis general:

```
expresiónBooleana else throw excepción
```

Ejemplo:

```
void Ejemplo(int numero)
{

  check(
    numero >= 0        else throw new NumeroNegativoException(),
    (numero % 2) == 0  else throw new NumeroNoParException(),
    AlgunMetodo(numero)
  );
}
```

Al igual que en la instrucción `checkis` las pruebas se realizan en estricto orden secuencial, es decir, si falla la primera prueba se lanza la excepción y no se ejecutan las siguientes, la segunda prueba se ejecuta solo si la primera pasó exitosamente; y así sucesivamente.

A.6.4 check estricto

Un `check` cuando las pruebas que contiene son:

- construcciones del tipo *expresiónBooleana* **else throw** *excepción*
- llamadas a chequeadores

Es decir, una prueba no puede ser simplemente una llamada a un método.

Ejemplo: El ejemplo anterior no es un `check` estricto ya que la última de sus pruebas es una llamada a un método.

A.7 Chequeadores

Son una unidad programática diseñada para probar el cumplimiento o no de una serie de condiciones en los elementos dados por argumento y que van enmarcadas dentro de construcción orientada a objetos.

A.7.1 Descripción

Los chequeadores realizan las pruebas que indican si los elementos dados por argumentos cumplen una condición y dependiendo de la llamada puede retornar un booleano que indica si se cumple o no, o puede iniciar una excepción personalizada en el caso de incumplimiento de estas.

Los chequeadores agrupan todo el código necesario para determinar si los argumentos cumplen la condición, permitiendo así su reutilización sin tener que reescribir el código.

A.7.2 Declaración

Se debe crear un contenedor para el chequeador, que podría ser una clase o una estructura. El contenedor puede implementar más interfaces y si es una clase podría tener también una clase base de la que hereda. Se sugiere el uso del sufijo `Checker` para nombrar los contenedores de chequeadores.

Ejemplo:

```
class MiChequeador {/*...*/}
static class MiChequeador {/*...*/}
struct MiChequeador {/*...*/}
```

Para dar soporte a los aspectos en el lenguaje se ha creado un nuevo tipo de miembro, de posible empleo en clases, estructuras e interfaces; así como los campos y los métodos son tipos de miembros utilizables en una clase, estructura o interfaz, los chequeadores también lo son y se pueden declarar en los mismos lugares en donde es posible declarar un método.

Sintaxis general:

```
checker(parámetros)
{
  /*...*/
}
```

La declaración del chequeador se hace empleando la palabra `checker` y dentro de paréntesis se listan los parámetros; la lista de parámetros siguen la estructura y reglas que rigen a en los métodos para declarar los parámetros que recibe, por lo que es válido el empleo del modificador `params`; aunque el empleo de los modificadores `ref` y `out` no está permitido ya que su uso hacer perder el sentido de un chequeador: comprobar el cumplimento de una condición.

Se puede hacer sobrecarga de chequeadores tal como ocurre en los métodos, ya que los parámetros forman parte de la firma del mismo.

A.7.2.1 Implementación

Dentro de las llaves de implementación del chequeador se hace todas las pruebas para comprobar que los parámetros cumplen con la condición, siguiendo la notación de un `check` estricto.

Las excepciones resultantes de la comprobación del `check` estricto son lanzadas solo en los casos en que el chequeador ha sido invocado de la forma en que

lanza la excepción, en el caso en que deba devolver un booleano no es iniciada la excepción.

Ejemplo:

```
static class EsParPositivoChecker
{

  public static checker(int numero)
  {

    numero >= 0      else throw new NumeroNegativoException(),
    (numero % 2)==0  else throw new NumeroNoParException()
  }
}
```

A.7.2.2 Implementación avanzada

En un chequeador es posible realizar una implementación personalizada para el caso en que daba devolver un booleano y el caso en que deba lanzar una excepción, para ello, en lugar de escribir la implementación del chequeador dentro de las llaves se escribe el descriptor `check` para la implementación en la que debe lanzar una excepción y el descriptor `checkis` para la implementación booleana.

Sintaxis general:

```
checker(parámetros)
{
  check { /*...*/ }
  checkis { /*...*/ }
}
```

En el caso del descriptor `checkis` la implementación debe retornar un valor booleano utilizando la sentencia `back` que indica si los parámetros pasan la prueba. En el caso del descriptor `check` deben iniciar una excepción que indica que los parámetros no pasan la prueba.

Ejemplo: Una reescritura del ejemplo anterior utilizando la implementación avanzada sería la siguiente

```
static class EsParPositivoChecker
{

  public static checker(int numero)
  {

    check {
      if (numero >= 0)
        if ( (numero % 2)==0 )
          back;
        else
          throw new NumeroNoParException();

      else
        throw new NumeroNegativoException();
    }

    checkis {
      if (numero >= 0)
        if ( (numero % 2)==0 )
          back true;
        else
          back false;

      else
        back fasle;
    }
  }
}
```

Importante: Un chequeador debe tener ambos descriptores.

A.7.2.3 Declaración sin implementación

En el caso de no dar implementación al chequeador se coloca punto y coma después de enunciar todos los parámetros.

Sintaxis general:

```
checker(parámetros);
```

Esta forma se utiliza para declarar chequeadores en interfaces o chequeadores que lleven el modificador `abstract`.

A.7.2.4 Implementación explícita

Para realizar una implementación explícita de un chequeador, se omiten los modificadores de este y se antepone a la palabra `checker` el nombre de la interfaz seguido de un punto.

Ejemplo:

```
IMiInterfaz.checker() {/* ... */}
```

A.7.2.5 Excepciones en la implementación básica

El la implementación básica de un chequeador, si algún método que es invocado lanza una excepción, esa excepción no es considerada un fallo de la prueba, por lo que para la versión booleana sería una excepción que atraviesa el chequeador; si se desea cambiar este comportamiento entonces se debe realizar la implementación avanzada del chequeador, desde donde se puede capturar la excepción e indicar si esto provoca el fallo de la prueba o su superación.

A.7.3 Utilización

Para invocar a un chequeador se utilizan las instrucciones de chequeo, primero se anuncia con la palabra `check` o `checkis`, luego se escribe en nombre de la clase contenedora, opcionalmente se puede escribir un punto, y por último y dentro de paréntesis se listan los argumentos que utiliza; la instrucción `check` sirve para invocar la implementación que lanza una excepción en caso de que falle la

prueba y la instrucción `checkis` para invocar la implementación que retorna un booleano.

Sintaxis general:

```
check NombreChequeador(argumentos)
checkis NombreChequeador(argumentos)
```

Los chequeadores se pueden utilizar en cualquier lugar en el que sea válido invocar un método, tales como métodos, propiedades, bloques de códigos y aspectos.

Ejemplo:

```
static class EsParPositivoChecker
{
  public static checker(int numero)
  {

    numero >= 0      else throw new NumeroNegativoException(),
    (numero % 2)==0  else throw new NumeroNoParException()
  }
}
```

Para llamar al descriptor `check`:

```
check EsParPositivoChecker(-2);
```

Al ejecutarse el chequeo de la llamada anterior es lanzada una excepción: `NumeroNegativoException`, pero si el argumento es 3 el método culminaría su ejecución sin iniciar ninguna excepción.

Para llamar al descriptor `checkis`:

```
bool resultado = checkis EsParPositivoChecker(-2);
```

Al ejecutarse el chequeo de la llamada anterior la variable resultado

contendrá el valor `false`, pero si el argumento es 3 el valor de la variable sería `true`.

Los chequeadores pueden ser utilizados en las instrucciones `check` y `checkis` empleando nuevamente la instrucción.

Ejemplo: Utilización de un chequeador en una instrucción `checkis`

```
bool Ejemplo(int numero)
{

   return checkis(
      numero > 0,
      checkis NumeroParChecker()
   );
}
```

Ejemplo: Utilización de un chequeador en la instrucción `check`

```
void Ejemplo(int numero)
{

   check(
      numero >= 0 else throw new NumeroNegativoException(),
      check NumeroParChecker()
   );
}
```

A.7.3.1 Chequeadores instanciados

Es posible utilizar chequeadores que para su uso es necesario que exista una instancia previa, para ello, después haber anunciado la utilización del chequeador con las palabras `check` o `checkis`, en vez de escribir el nombre de la clase contenedora del chequeador estático, se usa algunas de las siguientes construcciones:

- Para aplicar un chequeador instanciado previamente creado, se coloca el nombre de la variable que referencia a la instancia del contenedor del

chequeador y luego los argumentos que recibe el chequeador (opcionalmente se puede escribir un punto para separar a ambos).

Ejemplo:

```
void HacerAlgo(string s, ChequeadorNoEstatico a) {
  check a(s);
}


void HacerAlgo2(string s, OtroObjeto obj) {
  check obj.ChequeadorUsado(s);
{/*...*/}


void HacerAlgo3(string s, OtroObjeto obj)
  check obj.GetChequeadorUsado()(s);
{/*...*/}
```

- Para crear una instancia de un chequeador, se puede declara la variable que referenciará al contenedor del chequeador o (de ya estar declarada) simplemente escribir el nombre de la variable, luego se coloca los argumentos del chequeador (opcionalmente se puede escribir un punto para separar a ambos), posteriormente se escribe el signo = (igual) y se hace la inicialización de objeto contenedor del chequeador, bien sea llamando a un constructor o asignado un objeto preexistente.

  Ejemplo:

```
void HacerAlgo1(string s) {
  check ChequeadorNoEstatico a(s) =
    new ChequeadorNoEstatico();
}


void HacerAlgo2(string s, ChequeadorNoEstatico a) {
  check ChequeadorNoEstatico b(s) = a;
}


void HacerAlgo3(string s, OtroObjeto obj) {
  check ChequeadorNoEstatico b(s) =
```

```
    obj.GetChequeadorUsado();
}

void HacerAlgo4(string s, OtroObjeto obj)
  check ChequeadorNoEstatico b(s) = obj.ChequeadorUsado;
}
```

Nota: El ámbito de la variable declarada es el mismo que en su lugar se hubiera realizado una declaración de variable típica, se restringe al bloque de código actual.

- Se puede crear una instancia anónima del contenedor de un chequeador (creada in situ), para ello se crea la instancia de la clase contenedora del chequeador, luego se coloca los argumentos del chequeador (opcionalmente se puede escribir un punto para separar a ambos).

  Ejemplo:

```
void HacerAlgo(string s, OtroObjeto obj) {
  check new ChequeadorNoEstatico()(s);
}
```

A.7.4 check estricto y chequeadores

En un `check` estricto es posible llamar a un chequeador como una prueba en lugar de usar la construcción *`expresiónBooleana`* `else throw` *`excepción`*, para ello se invoca al chequeador haciendo uso de la instrucción `check`.

Ejemplo:

```
static class EsParPositivoChecker
{

  public static checker(int numero)
  {

    numero >= 0 else throw new NumeroNegativoException(),
```

```
        check EsParChecker(numero)
    }
}
```

La implementación del chequeador anterior es un `check` estricto ya que solo hace uso de la construcción *`expresiónBooleana`* `else throw` *`excepción`* y de la llamada al chequeador `EsParChecker`.

A.8 Contratos

Son una unidad programática diseñada para probar el cumplimiento o no de una serie de condiciones en las entradas y salidas de una unidad funcional y que van enmarcadas dentro de una construcción orientada a objetos.

A.8.1 Descripción

En una unidad funcional los contratos realizan comprobaciones iniciales y posteriores (en caso se ejecución normal) a la ejecución del mismo, estas comprobaciones se encuentran separadas del código y agrupadas en una construcción orientada a objetos, permitiendo al contrato ser reutilizable en diferentes unidades funcionales.

Un contrato se puede aplicar sobre una unidad funcional, ya sean métodos, propiedades o bloques de códigos.

Los contratos son similares a los aspectos, pero solo realizan comprobaciones antes de que se ejecute la primera instrucción de la unidad funcional sobre la cual se aplica y después de la culminación normal de la misma; no se realiza ningún tipo de acción en caso de que se una excepción atreviese; y a diferencia de los aspectos

los contratos se heredan, por lo que una reimplementación de un método debe cumplir los contratos a los que se encuentra sometido su antecesor.

A.8.2 Declaración

Se debe crear un contenedor para el contrato, que podría ser una clase o una estructura. El contenedor puede implementar más interfaces y si es una clase podría tener también una clase base de la que hereda. Se sugiere el uso del sufijo `Contract` para nombrar los contenedores de contratos.

Ejemplo:

```
class MiContrato {/*...*/}
static class MiContrato {/*...*/}
struct MiContrato {/*...*/}
```

Para dar soporte a los contratos en el lenguaje se ha creado un nuevo tipo de miembro, de posible empleo en clases, estructuras e interfaces; así como los campos y los métodos son tipos de miembros utilizables en una clase, estructura o interfaz, los contratos también lo son y se pueden declarar en los mismos lugares en donde es posible declarar un método.

Sintaxis general:

```
contract(parámetros de ingreso):(parámetros de egreso)
{
   require {/*...*/}
   ensure {/*...*/}
}
```

La declaración del contrato se hace empleando la palabra `contract` y dentro de paréntesis se listan los parámetros de ingreso del contrato, seguido de dos puntos y nuevamente dentro de paréntesis se listan los parámetros de egreso del

contrato; la lista de parámetros (ya sean de ingreso o de egreso) siguen la estructura y reglas que rigen a los métodos para declarar los parámetros que recibe, por lo que es válido el uso del modificador `params;` aunque el empleo de los modificadores `ref` y `out` no está permitido ya que su uso hacer perder el sentido de un contrato: comprobar el cumplimento de una serie de condiciones.

Se puede hacer sobrecarga de contratos tal como ocurre en los métodos, ya que los parámetros (de ingreso y de egreso) forman parte de la firma del mismo.

A.8.2.1 Los descriptores

El descriptor `requiere` indica las comporbaciones que se deben realizar antes de la ejecución de la primera instrucción del código sobre el cual se aplica el contrato.

El descriptor `ensure` indica las comprobaciones que se deben realizar después de la ejecución de la última instrucción del código sobre el cual se aplica el aspecto, siempre y cuando no sea iniciada una excepción.

Cada descriptor es per se un chequeado, por lo que su implementación se puede realizar siguiendo la implementación básica de chequeadores o se pueden colocar dentro del descriptor la implementación avanzada de chequeadores.

Nota: En un contrato no es necesario mencionar todos los descriptores.

A.8.2.2 Parámetros de ingreso

Son parámetros que existen al momento de entrar a la unidad funcional sobre la cual se aplica el contrato, es decir, antes de la ejecución de la primera instrucción del código sobre el cual se aplica el contrato deben poseer un valor; los parámetros

de ingreso son de paso por valor (por defecto) y pueden poseer una lista de argumentos variable (con el modificador `params`), como si de un método se tratase; no se permite el uso de los modificadores `ref` y `out`.

Los parámetros de ingreso están disponibles para su uso en cualquiera de los descriptores.

A.8.2.3 Parámetros de egreso

Son parámetros que existen al momento de salir de la unidad funcional sobre la cual se aplica el contrato, es decir, al momento de egresar de la unidad funcional deben poseer un valor; los parámetros de egreso son de paso por valor (por defecto) y pueden poseer una lista de argumentos variable (con el modificador `params`), como si de un método se tratase; no se permite el uso de los modificadores `ref` y `out`.

Los parámetros de egreso están disponibles para su uso en el descriptor `ensure`, pero no se pueden utilizar en el descriptor `require`.

A.8.2.4 Modificadores

Un contrato puede recibir los mismos modificadores que un método excepto el modificador `extern`, es decir, los modificadores válidos son: `public`, `protected`, `internal`, `private`, `static`, `virtual`, `sealed`, `new`, `override` y `abstract`, y su utilización se rige bajo las mismas reglas que deben seguirse en el caso de los métodos.

A.8.2.5 Implementación

Cada descriptor de contrato se implementa tal como si de un chequeador se

tratase, es decir, dentro de las llaves de implementación del descriptor del contrato se hace todas las pruebas para comprobar que los parámetros cumplen con la condición, siguiendo la notación de un `check` estricto.

Ejemplo: Un contrato que rige el calculo de raíces cuadradas (haciendo la salvedad de los errores que puedan ocurrir a causa del redondeo y la precisión finita del tipo de dato usado) podría ser el siguiente

```
static class RaizCuadradaContract
{

  public static contract(double argumento):(double resultado)
  {

    require {
      argumento >= 0 else throw new NumeroNegativoException()
    }

    ensure {
      argumento * argumento == resultado else throw
        new ResultadoErradoException(),
      resultado >= 0 else throw new ResultadoNegativoException()
    }
  }
}
```

Debido a que cada descriptor del contrato es un chequeador per se, es posible implementarlos utilizando la implementación avanzada de chequeadores, para ello hay que colocar los descriptores de la implementación avanzada del chequeador dentro del descriptor del contrato.

Ejemplo: implementación del descriptor `ensure` del ejemplo anterior utilizando la implementación avanzada de chequeadores.

```
static class RaizCuadradaContract
{

  public static contract(double argumento):(double resultado)
  {
```

```
        require {
          argumento >= 0 else throw new NumeroNegativoException()
        }

        ensure {
          check {
            if (argumento * argumento == resultado)
              if (resulatdo >=0)
                back;
              else
                throw new ResultadoNegativoException();

            else
              throw new ResultadoErradoException();
          }

          checkis {
            if (argumento * argumento == resultado)
              if (resulatdo >=0)
                back true;
              else
                back false;

            else
              back false;
          }
        } // fin del ensure
      } // fin de contract
    }
```

A.8.2.6 Declaración sin implementación

En el caso de no dar implementación a los descriptores se coloca punto y coma después de enunciar todos los parámetros.

Sintaxis general:

contract(*parámetros de ingreso***):(***parámetros de egreso***);**

Esta forma se usa para declarar aspectos en interfaces o aspectos que lleven el modificador `abstract`.

A.8.2.7 Implementación explícita

Para realizar una implementación explícita de un aspecto, se omiten los modificadores de este y se antepone a la palabra `contract` el nombre de la interfaz seguido de un punto.

Ejemplo:

```
IMiInterfaz.contract():() {/* ... */}
```

A.8.2.8 Asignaciones preliminares

En un contrato es posible realizar asignaciones preliminares a la ejecución del descriptor `require`, estas asignaciones se realizan en el mismo nivel en el que se coloca la implementación de los descriptores, es decir, se coloca dentro de las llaves de implementación del contrato, pero fuera de algún descriptor.

Nota: Estas asignaciones tienen el mismo efecto que si se hubiera hecho en las primeras líneas del descriptor `require` en su implementación avanzada (en ambos descriptores: `check` y `checkis`)

Ejemplo:

```
struct DepositoContract
{

  decimal balance_anterior;

  contract (decimal balance, decimal importe):()
  {
    balance_anterior = balance;

    require {
      importe >= 0 else throw
        new ArgumentOutOfRangeException(
          “importe”, “importe no puede ser negativo”
        )
    }
```

```
      ensure {
         balance == balance_anterior + importe else throw
           new Exception(
             “El resultado del método es errado”
           )
      }
   } // fin contract
}
```

A.8.3 Utilización

Los contratos se pueden utilizar en cualquier unidad funcional, tales como métodos, constructores, propiedades y bloques de códigos, su utilización se anuncia con la palabra `contract` y listando cada uno de los contratos por el nombre de la clase o estructura contenedora y dentro de los correspondientes paréntesis la lista de argumentos que requiere (opcionalmente se puede escribir un punto para separar a ambos), cada aspecto a utilizar se separa con coma, o se enuncia nuevamente con la palabra `contract`.

La utilización de los contratos es similar a la utilización de los aspectos, pero en lugar de escribir la palabra `aspect` para anunciar su aplicación se escribe la palabra `contract`, se puede aplicar contratos sobre métodos y descriptores de propiedades al igual que los aspectos.

Ejemplo:

```
void MiMetodo(int a, int b, int c, out int d)
   contract MiContrato.(a, b):(d), OtroContrato(b, c):(d)
{/*...*/}


void OtroMetodo(ref string s)
   contract UnContrato(s):()
   contract AlgunOtroContrato():()
{/*...*/}
```

```
int Propiedad
{

  get contract MiContrato():()
  {/*...*/}

  set contract MiContrato(value):()
  {/*...*/}
}
```

También se pueden aplicar contratos sobre descriptores de indizadores, sobrecarga de operadores, composición de contratos y aplicación de contratos sobre bloques de códigos; todo esto es análogo a los aspectos haciendo la salvedad que la utilización de los contratos se anuncia con la palabra `contract`.

Ejemplo: Contratos compuestos

```
contract(int parametro):()
  contract MiContrato(parametro, “Algún valor”):()
{
  require {/*...*/}
  ensure {/*...*/}
}
```

Ejemplo: Contratos sobre bloques de códigos

```
{
  int x = 0, y;

  using
    contract UnContrato(x):(y)
  {
    otrasOpertaciones();
    // ...
    y = 3;
  }

  // ...

  using (MiObjeto z = new MiObjeto)
    contract OtroContrato(x,y):(z)
  { /* ... */ }
}
```

A.8.3.1 Invocar un descriptor

Debido a que cada descriptor de un contrato es un chequeador, se pueden invocar como tal, para comprobar el cumplimiento o no de cada descriptor del contrato por separado. Para ello se hace uso de la misma instrucciones empleadas para la invocación chequeadores, pero después de haber escrito la palabra `check` o `checkis` se escribe la palabra `require` o `ensure` según sea el descriptor del contrato que se desee invocar, luego se indica el nombre del contrato en cuestión y sus argumentos (de tratarse de un contrato estático, también se puede emplear cualquiera de las variaciones para la invocación de contratos instanciados).

Ejemplo:

```
static class RaizCuadradaContract
{

   public static contract(double argumento):(double resultado)
   {

      require {
         argumento >= 0 else throw new NumeroNegativoException()
      }

      ensure {
         argumento * argumento == resultado else throw
            new ResultadoErradoException(),
         resultado >= 0 else throw new ResultadoNegativoException()
      }
   }
}
```

La invocación de la versión booleana o la que inicia una excepción para cada uno de los descriptores del contrato sería como:

Versión que inicia una excepción del descriptor `require` del contrato

```
   check require RaizCuadradaContract(4d):(0d);
```

Versión booleana del descriptor `require del contrato`

```
checkis require RaizCuadradaContract(4d):(0d);
```

Versión que inicia una excepción del descriptor `ensure` del contrato

```
check ensure RaizCuadradaContract(4d):(2d);
```

Versión booleana del descriptor `ensure` del contrato

```
checkis ensure RaizCuadradaContract(4d):(2d);
```

Nota: a pesar de que el descriptor `require` no tiene acceso a los parámetros de egreso es necesario indicarlos, y sin importar los valores que se le asignen el resultado es el mismo, ya que estos parámetros no afectan la ejecución del descriptor, pero son necesario indicarlos para saber de cuál sobrecarga del contrato se trata.

A.8.4 Orden de aplicación

Las consideraciones respecto al orden de aplicación de los contratos son las mismas que para los aspectos, es decir, el orden de ejecución del descriptor `require` es el mismo que el orden en que se enunciaron los contratos, pero para el descriptor `ensure` el orden es inverso; por los que al aplicar varios contratos es conveniente ordenarlos del más general al más específico.

Ejemplo:

```
void MiMetodo()
  contract Contrato1():()
  contract Contrato2():()
  contract Contrato3():()
{
  // ... Contenido del método ...
```

```
}
```

Al invocar a `MiMetodo` se ejecutará primero el descriptor `require` de `Contrato1`, luego el de `Contrato2`, y por último el de `Contrato3`; luego se ejecuta el contenido del método, de no haber sido lanzada una excepción se ejecuta el descriptor `ensure` de `Contrato3`, luego de no haber sido lanzada una excepción se ejecuta el descriptor `ensure` de `Contrato2` y por último de no haber sido lanzada una excepción se ejecuta el descriptor `ensure` de `Contrato1`.

El resultado es similar de haber escrito:

```
void MiMetodo() {
  using contract Contrato1():() {
    using contract Contrato2():() {
      using contract Contrato3():() {
        // ... Contenido del método ...
      }
    }
  }
}
```

A.8.5 Contratos instanciados

En los contratos al igual que en los aspectos es posible utilizar contratos que para su empleo es necesario que exista una instancia previa, las reglas y construcciones para utilizarlos son las mismas que para los aspectos.

Nota: al igual que en los aspectos la únicas construcciones válidas para ser utilizadas en la composición de contratos son aquellas que emplean contratos instanciados previamente creados.

Ejemplo: ejemplos donde se utilizan las construcciones válidas para la composición de contratos.

```
void HacerAlgo(string s, ContratoNoEstatico a)
  contract a(s):()
{/*...*/}


void HacerAlgo2(string s, OtroObjeto obj)
  contract obj.ContratoUsado(s):()
{/*...*/}


void HacerAlgo3(string s, OtroObjeto obj)
  contract obj.GetContratoUsado()(s):()
{/*...*/}
```

Ejemplo: ejemplos donde se utilizan las construcciones no válidas para la composición de contratos.

```
void HacerAlgo4(string s)
  contract ContratoNoEstatico a(s):() =
    new ContratoNoEstatico()
{/*...*/}


void HacerAlgo5(string s, ContratoNoEstatico a)
  contract ContratoNoEstatico b(s):() = a
{/*...*/}


void HacerAlgo6(string s, OtroObjeto obj)
  contract ContratoNoEstatico b(s):() =
    obj.GetContratoUsado()
{/*...*/}


void HacerAlgo7(string s, OtroObjeto obj)
  contract ContratoNoEstatico b(s):() = obj.ContratoUsado
{/*...*/}


void HacerAlgo8(string s, OtroObjeto obj)
  contract new ContratoNoEstatico()(s):()
{/*...*/}


void MiMetodo(ContratoNoEstatico a)
  contract a():(), new OtroContratoNoEstatico()():()
{/*...*/}
```

A.8.6 El valor de retorno

Al igual que en los aspectos los contratos pueden acceder al valor retornado por un método o una propiedad `get`, este se puede obtener empleando la palabra `returned`. Esto solo aplica en los contratos aplicados directamente sobre métodos y propiedades `get` (aplicados antes de la apertura de la apertura de la llave de implementación), no sobre bloques de códigos en general.

Ejemplo:

```
int UnMetodo(string s)
  contract MiContrato(s):(returned)
{/*...*/}


int OtroMetodo(string s, ContratoNoEstatico a)
  contract a(s):(returned)
{/*...*/}


int Propiedad
{
  get contract MiContrato( ):(returned) {/*...*/}
  set contract MiContrato(value):( ) {/*...*/}
}
```

A.8.7 Contratos compuestos

Al igual que los aspectos, los contratos se pueden componer de otros contratos, de forma tal que la funcionalidad de este sea la combinación de la funcionalidad de los otros contratos que lo componen y la funcionalidad propia.

Ejemplo:

Si se tiene tres contratos. `A`, `B` y `C`, y el contrato `C` se compone con los contratos `A` y `B`:

```
static class A {

  public static contract():() {

    require { PruebaRA else throw ExcepcionRA }
    ensure { PruebaEA else throw ExcepcionEA }
  }
}


static class B {

  public static contract():() {

    require { PruebaRB else throw ExcepcionRB }
    ensure { PruebaEB else throw ExcepcionEB }
  }
}


static class C {

  public static contract():()
    contract A():()
    contract B():()
  {

    require { PruebaRC else throw ExcepcionRC }
    ensure { PruebaEC else throw ExcepcionEC }
  }
}
```

El comportamiento del contrato `C` es que para cada uno de los descriptores se realiza la acción de cada descriptor equivalente según el orden de aparición para el descriptor `require` y en orden inverso para `ensure`, por lo que un equivalente funcional a `C` sería:

```
static class C {

  public static contract():() {

    require {
      PruebaRA else throw ExcepcionRA,
      PruebaRB else throw ExcepcionRB,
      PruebaRC else throw ExcepcionRC
    }

    ensure {
      PruebaRC else throw ExcepcionRC,
      PruebaRB else throw ExcepcionRB,
```

```
            PruebaRA else throw ExcepcionRA
        }
    }
}
```

Todas las reglas y consideraciones en la composición de aspectos aplican en la composición de contratos: el orden de composición, la forma de alterar el orden de composición (empleando la palabra `here`), la composición recursiva y la composición con contratos sobrecargados.

A.8.8 Sobrecarga de contratos

Al igual que los aspectos, los contratos son sobrecargables en función de su lista de parámetros de ingreso y egreso; tal como ocurre en los métodos, ya que los parámetros (de ingreso y de egreso) forman parte de la firma del mismo.

Los contratos sobrecargados también se pueden intercomponer al igual que los aspectos, siguiendo las mismas reglas y consideraciones que tienen estos últimos.

A.8.9 Contratos y herencia

Los contratos aplicados sobre un miembro de clase o estructura, tal como un método o una propiedad, que posea el modificador `virtual` o `abstaract` se heredan, es decir, cada una de las implementaciones posteriores se siguen rigiendo por el contrato aun cuando en la implementación posterior no se haya aplicado ese contrato.

Ejemplo:

```
static class FactorialContract
```

```
{

   public static contract(long argumento):(long resultado)
   {

      require {
         argumento >= 0 else throw new NumeroNegativoException()
      }

      ensure {
         resultado >= argumento else throw
            new ResultadoErradoException(),
         resultado >= 0 else throw new ResultadoNegativoException()
      }
   }
}


class MatematicaBase
{

   public virtual long Factorial(long x)
      contract FactorialContract(x):(returned)
   {
      if (x <=1)
         return 1;
      else
         return x * Factorial(x-1);
   }
}


class MatematicaHija : MatematicaBase
{

   public override long Factorial(long x)
   {
      long tmp = 1;
      for(long i=2; i<=x; i++)
         tmp *= i;
   }
}
```

La implementación del método Factorial en la clase `MatematicaHija` está sujeta al contrato `FactorialContract`, aplicado en la case base; por lo que si se llama a

```
(new MatematicaHija()).Factorial(-5)
```

el resultado es que se iniciaría un `NumeroNegativoException` proveniente del descriptor `require` del contrato que rige a ese método, lo mismo ocurriría si se llamara a

```
(new MatematicaBase()).Factorial(-5)
```

A.8.9.1 Contratos heredados y nuevos contratos

Los métodos y propiedades que heredan un contrato no pueden recibir más contratos, si se desea cambiar las clausulas del contrato que rige al método o propiedad se debe hacer una ocultación del miembro heredado usando el modificar `new` e implementar el miembro con el nuevo contrato.

A.8.9.2 Contratos y miembros abstractos

Es posible aplicar contratos sobre miembros abstractos, tales como métodos o propiedades que posean el modificador `abstract`, a pesar de que no posean una implementación; los contratos aplicados sobre estos miembros son heredados por sus implementaciones.

Para aplicar un contrato sobre un miembro abstracto se siguen las mismas reglas para la aplicación de contratos, pero, en lugar de escribir las llaves de implementación del miembro se coloca punto y coma.

Ejemplo:

```
class MatematicaAbstracta
{

  public abstract long Factorial(long x)
    contract FactorialContract(x):(returned);

  public abstract double PI
  {
```

```
      get contract OtroContrato():(returned);
   }
}
```

A.8.9.3 Contratos e interfaces

Al igual que los métodos y propiedades con el modificador `abstract`, a los miembros declarados en una interfaz, tales como métodos o propiedades, se le pueden aplicar contratos; las implementaciones de esos miembros heredan esos contratos, por lo que se rigen por ellos.

Ejemplo:

```
interface IMatematica
{

   long Factorial(long x)
      contract FactorialContract(x):(returned);
}


class Matematica : IMatematica
{

   public virtual long Factorial(long x)
   {
      if (x <=1)
         return 1;
      else
         return x * Factorial(x-1);
   }
}
```

La implementación del método Factorial en la clase `Matematica` está sujeta al contrato `FactorialContract`, aplicado en la interfaz que implementa; por lo que si se llama a

```
(new Matematica()).Factorial(-5)
```

el resultado es que se iniciaría un `NumeroNegativoException` proveniente del

descriptor `require` del contrato que rige a ese método.

A.8.9.4 Contratos y la ocultación de miembros

Cuando se hace una ocultación de miembro empleando el modificador `new` el nuevo miembro no está sometido a las reglas del contrato que regía a su predecesor.

Ejemplo:

```
class MatematicaBase
{

  public virtual long Factorial(long x)
    contract FactorialContract(x):(returned)
  {
    if (x <=1)
      return 1;
    else
      return x * Factorial(x-1);
  }
}


class MatematicaHija : MatematicaBase
{

  public new long Factorial(long x)
  {
    return -100;
  }
}
```

La implementación del método Factorial en la clase `MatematicaHija` no está sujeta al contrato `FactorialContract`, aplicado en la case base; por lo que si se llama a

```
(new MatematicaHija()).Factorial(-5)
```

el resultado es `-100`, pero si se llama a alguna de las siguientes

```
(new MatematicaBase()).Factorial(-5)
```

```
( (MatematicaBase)(new MatematicaHija()) ).Factorial(-5)
```

que se iniciaría un `NumeroNegativoException` proveniente del descriptor `require` del contrato que rige a ese método en la clase base.

A.8.10 Contratos y aspectos

Cuando se combina la aplicación de contratos y aspectos sobre un mismo elemento es necesario que dicha aplicación se haga estrictamente en el siguiente orden: primero contratos, segundo aspectos; no está permitido ninguna otra organización, tales como intercalar aspectos y contratos o aplicar primero aspectos y luego contratos.

Ejemplo:

```
bool Depositar(decimal importe)
   contract new DepositoContract()(balance, importe):()
   aspect TransaccionAspect():()
{
   AgregarDeposito(importe);
}
```

A.8.10.1 Contratos como aspectos

Se puede aplicar un contrato como si de un aspecto se tratase, para ello se anuncia la aplicación de un aspecto, luego se escribe la palabra `contract` y se aplica el contrato.

Ejemplo:

```
void MiMetodo
   aspect contract MiContrato():()
{
   // ...
}
```

A.9 Contratos anónimos

Son contratos cuya implementación se realiza in situ, en el lugar donde se emplea. Los contratos anónimos carecen de contenedor, tienen la capacidad de capturar las variables externas.

A.9.1 Descripción

Con contratos anónimos es posible crear y utilizar contratos implementados in situ, sin tener que crear un contenedor y una implementación separada a su utilización. Los contratos anónimos tienen pleno acceso a las variables externas en sus descriptores.

A.9.2 Declaración

Un contrato anónimo se pueden emplear en los mismos lugares donde es aplicable un contrato, pero, en vez de indicar el nombre de la clase contenedora y los argumentos que recibe este, se coloca dentro de llaves la implementación de los descriptores del contrato.

Ejemplo:

```
class MatematicaBase
{

   public virtual long Factorial(long x)
      contract {
         require {
            x >= 0 else throw new NumeroNegativoException()
         }

         ensure {
            returned >= x else throw
               new ResultadoErradoException(),
```

```
            returned >= 0 else throw
              new ResultadoNegativoException()
          }
        }
      {
        if (x <=1)
          return 1;
        else
          return x * Factorial(x-1);
      }
    }
```

Nota: Al utilizar la implementación avanzada de chequeadores en contratos anónimos no es requerida la implementación del descriptor `checkis` ya que no se utiliza.

A.9.3 Utilización

Se puede utilizar contratos anónimos en los mismos lugares en los que se puede aplicar un contrato, y en su implementación se tiene acceso a todos los elementos que a ese nivel se tiene visibilidad; si se aplica sobre o dentro de un método o propiedad, se tiene acceso a los parámetros de éste.

A.9.4 Variables internas

En los contratos anónimos es posible declarar variables internas al contrato, a las que el elemento sobre el cual se aplica no tiene visibilidad, pero que dichas variables mantienen su valor hasta haber culminado la ejecución del descriptor `ensure`.

Nota: No tiene sentido declarar una variable interna si el contrato anónimo no posee un descriptor `ensure` en su implementación.

Las variables internas de contratos anónimos extienden las capacidades de las

asignaciones preliminares de los contratos, permitiendo declarar variables. Las variables internas de contratos anónimos se declaran y opcionalmente se inicializan en el mismo nivel en el que se coloca la implementación de los descriptores, es decir, se coloca dentro de las llaves de implementación del contrato, pero fuera de algún descriptor.

Nota: Declarar e inicializar en la misma sentencia tiene el mismo efecto que si se hubiera declarado la variable interna y en las primeras líneas del descriptor `require` en su implementación avanzada se hubiera inicializado.

Ejemplo: Supóngase una clase que representa una cuenta bancaria que cuenta con un método `Depositar` que recibe un importe como argumento y agrega ese importe al saldo

```
bool Depositar(decimal importe)
  contract
  {

    decimal balance_anterior = balance;

    require {
      importe >= 0 else throw
        new ArgumentOutOfRangeException(
          “importe”, “importe no puede ser negativo”
        )
    }

    ensure {
      balance == balance_anterior + importe else throw
        new Exception(
          “El resultado del método es errado”
        )
    }
  } // fin contract

{
  AgregarDeposito(importe);
}
```

Otra forma de haber resuelto este problema es haber utilizado un contrato

instanciado de inspección contenido en una estructura con un campo que equivaldría a la variable interna. Esta segunda forma permite la reutilización del contrato.

De esta otra forma el método `Depositar` queda como:

```
bool Depositar(decimal importe)
  contract new DepositoContract()(balance, importe):()
{
  AgregarDeposito(importe);
}
```

Y el contrato utilizado es:

```
struct DepositoContract
{

  decimal balance_anterior;

  contract (decimal balance, decimal importe):()
  {
    balance_anterior = balance;

    require {
      importe >= 0 else throw
        new ArgumentOutOfRangeException(
          “importe”, “importe no puede ser negativo”
        )
    }


    ensure {
      balance == balance_anterior + importe else throw
        new Exception(
          “El resultado del método es errado”
        )
    }
  } // fin contract
}
```

A.10 Contratos desmembrados

Son contratos cuya implementación de sus descriptores se hace in situ, en el

lugar donde se emplea, no requieren de definir el contrato como tal y mucho menos un contenedor. En estos descriptores se puede capturar las variables externas y hacer uso de ellas.

A.10.1 Descripción

Con los contratos desmembrados es posible implementar in situ el descriptor de un contrato, si tener que crear un contrato y mucho menos un contenedor para este. Estos descriptores tiene pleno acceso a las variables externas.

A.10.2 Declaración

Para emplear un contrato desmembrado, en vez de anunciar la aplicación de un contrato con la palabra `contract`, se coloca el nombre del descriptor y dentro de llaves la implementación del mismo. Al igual que en la aplicación de contratos, se pueden separar varias implementaciones del descriptor con coma; o se puede enunciar el descriptor nuevamente y realizar otra implementación que complementa a la primera.

Ejemplo: Utilizando contratos desmembrados en el ejemplo de cálculo de un factorial

```
public long Factorial(long x)
   require {
      x >= 0 else throw new NumeroNegativoException()
   }
   ensure {
      returned >= x else throw
         new ResultadoErradoException(),
      returned >= 0 else throw
         new ResultadoNegativoException()
   }
{
   if (x <=1)
```

```
    return 1;
  else
    return x * Factorial(x-1);
}
```

Nota: Al utilizar la implementación avanzada de chequeadores en contratos desmembrados no es requerida la implementación del descriptor `checkis` ya que no se utiliza.

A.10.3 Utilización

Se puede emplear contratos desmembrados en los mismos lugares en los que se puede aplicar un contrato, y en su implementación se tiene acceso a todos los elementos que a ese nivel se tiene visibilidad; si se aplica sobre o dentro de un método o propiedad, se tiene acceso a los parámetros de éste.

A.10.4 Orden de aplicación

El orden de aplicación sigue las mismas reglas de orden de aplicación de contratos, el orden de ejecución del descriptor `require` es el mismo que el orden en que se enunciaron (bien sea dentro de un contrato, o porque se enunció el descriptor), pero para el descriptor `ensure` el orden es inverso; por los que es conveniente ordenarlos del más general al más específico.

Ejemplo:

```
void MiMetodo()
  contract Contrato1():()
  require { /* require1 */ }
  ensure { /* ensuret1 */ }
  contract {require{/*require2*/} ensure{/*ensure2*/}},
    Contrato2():()

{
```

```
  // ... Contenido del método ...
}
```

El resultado es similar de haber escrito (haciendo la salvedad de lo referente a la herencia de contratos):

```
void MiMetodo()
{

  using contract Contract1():() {
    using require { /* require1 */ } {
      using ensure { /* ensure1 */ } {
        using contract {
          require{/*require2*/}
          ensure{/*ensure2*/} }
        {
          using Contrato2():() {
            // ... Contenido del método ...
          }
        }
      } // fin de ensure1
    } // fin de require1
  } // fin de Contrato1():()
}
```

A.10.5 Fortificación de contratos heredados

Los métodos y propiedades que heredan un contrato no pueden recibir más contratos, pero sí pueden fortificar las postcondiciones, es decir, agregar más postcondiciones. Las fortificaciones de las postcondiciones se hacen a través de contratos desmembrados, aplicando el descriptor `ensure` sobre el elemento a fortificar.

Ejemplo:

```
static class FactorialContract
{

  public static contract(long argumento):(long resultado)
  {
```

```
      require {
        argumento >= 0 else throw new NumeroNegativoException()
      }

      ensure {
        resultado >= argumento else throw
          new ResultadoErradoException()
      }
    }
}


class MatematicaBase
{

  public virtual long Factorial(long x)
    contract FactorialContract(x):(returned)
  {
    if (x <=1)
      return 1;
    else
      return x * Factorial(x-1);
  }
}


class MatematicaHija : MatematicaBase
{

  public override long Factorial(long x)
    ensure {
      resultado >= 0 else throw new ResultadoNegativoException()
    }
  {
    long tmp = 1;
    for(long i=2; i<=x; i++)
      tmp *= i;
  }
}
```

A.10.5.1 Orden de aplicación

Al fortificar un contrato, el orden de aplicación de las clausulas `ensure` heredadas y las clausulas fortificantes es la siguiente: Primero se chequean las clausulas heredadas y si estas son satisfechas se continúa con las nuevas clausulas que fortifican el contrato.

A.11 Propiedades envolventes

Las propiedades envolventes son propiedades que permiten autocontener el campo que las soporta, donde la accesibilidad a ese campo está restringida exclusivamente a la propiedad que lo envuelve; para dar soporte a las propiedades envolventes es necesario permitir que la declaración (e incluso inicialización) del campo se encuentre dentro de la propiedad que lo envuelve, al mismo nivel en el que se encuentran los descriptores.

Nota: una propiedad envolvente puede contener la declaración de más de un campo.

Importante: el identificar del campo sigue las mismas reglas de un campo tradicional, por lo que debe ser único dentro de la clase o estructura.

Ejemplo: Ninguna unidad funcional puede acceder al campo `_saldo` excepto la propiedad `Saldo`.

```
class CuentaBancaria {

  public decimal Saldo {
    decimal _saldo;
    get {
      return _saldo;
    }
    set {
      _saldo = value;
    }
  }
}
```

Una propiedad envolvente es una extensión a las propiedades tradicionales; por lo que al igual que estas últimas, sobre las propiedades envolventes es posible aplicar aspectos y contratos.

A.11.1 Propiedades envolventes como alternativa a invariantes

Se puede emplear una propiedad envolvente sobre el campo que es motivo de la invariante para evitar crear esta última, colocando las condiciones de la invariante al menos en la clausula `require` del contrato del descriptor `set` (anticipándose al cambio que haría perder la consistencia) que rige a la propiedad.

Ejemplo: (Se está utilizando la notación definida en el Apéndice E: Extensión para el soporte de invariantes para representar la invariante de clase)

```
class CuentaBancaria
{
  decimal _saldo;

  public decimal Saldo {
    get {
      return _saldo;
    }
    set {
      _saldo = value;
    }
  }

  public invariant SaldoPositivoInvariant {
    _saldo >= 0 else throw new SaldoIncorecctoException();
  }
}
```

La clase anterior se podría haber evitado utilizar la invariante empleando una propiedad envolvente, lo que sería:

```
class CuentaBancaria
{
  public decimal Saldo
  {
    decimal _saldo;
    get {
      return _saldo;
    }
    set
      require {
        value >=0 else throw
```

```
                new SaldoIncorecctoException()
            }
        {
            _saldo = value;
        }
    }
}
```

Importante: esta alternativa no es necesariamente correcta en el caso ser utilizada en objetos concurrentes.

Apéndice B

Gramática de C#

A continuación se presenta la la gramática léxica y sintáctica del lenguaje de programación C# definidas por la ECMA en ECMA-334 3ra edición en su Anexo A, también se presenta una explicación de la notación allí utilizada y respetada en este documento.

B.1 Convenciones de notación

La gramática léxica define cómo los caracteres se pueden combinar para formar lexemas (tokens), los elementos léxicos mínimos del lenguaje. La gramática sintáctica define como los lexemas (tokens) pueden ser combinados para hacer programas C# válidos.

Las producciones gramáticas incluyen símbolos tanto no-terminales como terminales. En producciones gramaticales, los símbolos *`no-terminales`* se muestran en letra cursiva, y los símbolos **`terminales`** se muestran en letra de ancho fijo. Cada no-terminal es definido por un conjunto de producciones. La primera línea de un

conjunto de producciones es el nombre del no-terminal, seguido de uno o un par de dos puntos. Dos puntos son utilizados son utilizadas en las producciones de la gramática sintáctica, un par de dos puntos son utilizadas en las producciones de la gramática léxica. Cada línea indentada sucesiva contiene el lado derecho para una producción que tiene el símbolo no-terminal a la izquierda. Por ejemplo:

```
class-modifier:
  new
  public
  protected
  internal
  private
  abstract
  sealed
  static
```

Las alternativas se listan normalmente en líneas separadas, como se muestra arriba, aunque en casos en los que haya varias alternativas, la frase "una de" precede la lista de opciones. Esto es simplemente una forma corta de listar cada una de las alternativas en líneas separadas. Por ejemplo:

```
decimal-digit: una de
  0   1   2   3   4   5   6   7   8   9
```

es equivalente a:

```
decimal-digit:
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
```

Un sufijo "$_{opc}$", como en *identificador*$_{opc}$, se utiliza como una forma corta de indicar un símbolo opcional. El ejemplo:

```
for-statement:
  for ( for-initializeropc ; for-conditionopc ; for-iteratoropc )
       embedded-statement
```

es equivalente a:

```
for-statement:
  for ( ; ; ) embedded-statement

for-statement:
  for ( for-initializer ; ; ) embedded-statement

for-statement:
  for ( ; for-condition ; ) embedded-statement

for-statement:
  for ( ; ; for-iterator ) embedded-statement

for-statement:
  for ( for-initializer ; for-condition ; ) embedded-statement

for-statement:
  for ( ; for-condition ; for-iterator ) embedded-statement

for-statement:
  for ( for-initializer ; ; for-iterator ) embedded-statement

for-statement:
  for ( for-initializer ; for-condition ; for-iterator )
       embedded-statement
```

Todos los caracteres terminales deben de ser entendidos como los caracteres Unicode apropiados del rango **U+0020** a **U+007F**, y no como caracteres similares en otros rangos Unicode.

B.2 Gramática léxica

```
input::
  input-sectionopc
```

input-section::
  input-section-part
  input-section input-section-part

input-section-part::
  input-elements$_{opc}$ *new-line*
  pp-directive

input-elements::
  input-element
  input-elements input-element

input-element::
  whitespace
  comment
  token

B.2.1 Terminadores de línea

new-line::
  Carácter de retorno de línea (**U+000D**)
  Carácter de alimentación de línea (**U+000A**)
  Carácter de retorno de línea (**U+000D**) seguido por un carácter
       alimentador de línea (**U+000A**)
  Carácter de siguiente línea (**U+2085**)
  Carácter separador de línea (**U+2028**)
  Carácter separador de párrafo (**U+2029**)

B.2.2 Espacio en blanco

whitespace::
  whitespace-characters

whitespace-characters::
  whitespace-character
  whitespace-characters whitespace-character

whitespace-character::
  Cualquier carácter de la clase Unicode Zs
  Carácter de tabulador horizontal (**U+0009**)
  Carácter de tabulador vertical (**U+000B**)
  Carácter alimentador de forma (**U+000C**)

B.2.3 Comentarios

comment::
  single-line-comment
  delimited-comment

single-line-comment::

```
    // input-characters_opc

input-characters::
    input-character
    input-characters input-character

input-character::
    Cualquier carácter Unicode excepto new-line-character

new-line-character::
    Carácter de retorno de línea (U+000D)
    Carácter de alimentación de línea (U+000A)
    Carácter separador de línea (U+2028)
    Carácter separador de párrafo (U+2029)

delimited-comment::
    /* delimited-comment-text_opc asterisks /

delimited-comment-text::
    delimited-comment-section
    delimited-comment-text delimited-comment-section

delimited-comment-section::
    not-asterisk
    asterisks not-slash

asterisks::
    *
    asterisks *

not-asterisk::
    Cualquier carácter Unicode excepto *

not-slash::
    Cualquier carácter Unicode excepto /
```

B.2.4 Símbolos

```
token::
    identifier
    keyword
    integer-literal
    real-literal
    character-literal
    string-literal
    operator-or-punctuator
```

B.2.5 Secuencias de escape Unicode

```
unicode-escape-sequence::
    \u hex-digit hex-digit hex-digit hex-digit
    \U hex-digit hex-digit hex-digit hex-digit hex-digit
```

hex-digit hex-digit hex-digit

B.2.6 Identificadores

identifier::
  available-identifier
  @ *identifier-or-keyword*

available-identifier::
  Un *identifier-or-keyword* que no es un *keyword*

identifier-or-keyword::
  identifier-start-character identifier-part-characters$_{opc}$

identifier-start-character::
  letter-character
  _ (el carácter guión-bajo **U+005F**)

identifier-part-characters::
  identifier-part-character
  identifier-part-characters identifier-part-character

identifier-part-character::
  letter-character
  decimal-digit-character
  connecting-character
  combining-character
  formatting-character

letter-character::
  Un carácter Unicode de clases Lu, Ll, Lt, Lm, Lo, o Nl
  Un *unicode-escape-sequence* representando un carácter de las clases Lu, Ll, Lt, Lm, Lo, o Nl

combining-character::
  Un carácter Unicode de las clases Mn o Mc
  Un *unicode-escape-sequence* representando un carácter de las clases Mn o Mc

decimal-digit-character::
  Un carácter Unicode de la case Nd
  Un *unicode-escape-sequence* representando un carácter de la clase Nd

connecting-character::
  Un carácter Unicode de la case Pc
  Un *unicode-escape-sequence* representando un carácter de la clase Pc

formatting-character::
  Un carácter Unicode de la case Cf
  Un *unicode-escape-sequence* representando un carácter de la clase Cf

B.2.7 Palabras clave

keyword:: una de

abstract	**as**	**base**	**bool**	**break**
byte	**case**	**catch**	**char**	**checked**
class	**const**	**continue**	**decimal**	**default**
delegate	**do**	**double**	**else**	**enum**
event	**explicit**	**extern**	**false**	**finally**
fixed	**float**	**for**	**foreach**	**goto**
if	**implicit**	**in**	**int**	**interface**
internal	**is**	**lock**	**long**	**namespace**
new	**null**	**object**	**operator**	**out**
override	**params**	**private**	**protected**	**public**
readonly	**ref**	**return**	**sbyte**	**sealed**
short	**sizeof**	**stackalloc**	**static**	**string**
struct	**switch**	**this**	**throw**	**true**
try	**typeof**	**uint**	**ulong**	**unchecked**
unsafe	**ushort**	**using**	**virtual**	**void**
volatile	**while**			

B.2.8 Literales

literal::
  boolean-literal
  integer-literal
  real-literal
  character-literal
  string-literal
  null-literal

boolean-literal::
  true
  false

integer-literal::
  decimal-integer-literal
  hexadecimal-integer-literal

decimal-integer-literal::
  decimal-digits integer-type-suffix$_{opc}$

decimal-digits::
  decimal-digit
  decimal-digits decimal-digit

decimal-digit:: una de
  0 **1** **2** **3** **4** **5** **6** **7** **8** **9**

integer-type-suffix:: una de
  U **u** **L** **l** **UL** **Ul** **uL** **ul** **LU** **Lu** **lU** **lu**

hexadecimal-integer-literal::
  0x *hex-digits integer-type-suffix*$_{opc}$
  0X *hex-digits integer-type-suffix*$_{opc}$

hex-digits::
  hex-digit
  hex-digits hex-digit

hex-digit:: una de
  0 **1** **2** **3** **4** **5** **6** **7** **8** **9**
  A **B** **C** **D** **E** **F**
  a **b** **c** **d** **e** **f**

real-literal::
  decimal-digits **.** *decimal-digits exponent-part*$_{opc}$
      real-type-suffix$_{opc}$
  . *decimal-digits exponent-part*$_{opc}$ *real-type-suffix*$_{opc}$
  decimal-digits exponent-part real-type-suffix$_{opc}$
  decimal-digits real-type-suffix

exponent-part::
  e *sign*$_{opc}$ *decimal-digits*
  E *sign*$_{opc}$ *decimal-digits*

sign:: una de
  + **-**

real-type-suffix:: una de
  F **f** **D** **d** **M** **m**

character-literal::
  ' *character* **'**

character::
  single-character
  simple-escape-sequence
  hexadecimal-escape-sequence
  unicode-escape-sequence

single-character::
  Cualquier carácter excepto **'** *(***U+0027***)*, **** *(***U+005C***)*, y
      new-line-character

simple-escape-sequence:: una de
  \' **\"** **** **\0** **\a** **\b** **\f** **\n** **\r** **\t** **\v**

hexadecimal-escape-sequence::
  \x *hex-digit hex-digit*$_{opc}$ *hex-digit*$_{opc}$ *hex-digit*$_{opc}$

string-literal::
  regular-string-literal
  verbatim-string-literal

regular-string-literal::
  " *regular-string-literal-characters*$_{opc}$ **"**

regular-string-literal-characters::

regular-string-literal-character
regular-string-literal-characters
regular-string-literal-character

regular-string-literal-character::
single-regular-string-literal-character
simple-escape-sequence
hexadecimal-escape-sequence
unicode-escape-sequence

single-regular-string-literal-character::
Cualquier carácter excepto **"** *(***U+0022***)*, **** *(***U+005C***)*, y
new-line-character

verbatim-string-literal::
@" *verbatim-string-literal-characters*$_{opc}$ **"**

verbatim-string-literal-characters::
verbatim-string-literal-character
verbatim-string-literal-characters
verbatim-string-literal-character

verbatim-string-literal-character::
single-verbatim-string-literal-character
quote-escape-sequence

single-verbatim-string-literal-character::
Cualquier carácter excepto **"**

quote-escape-sequence::
""

null-literal::
null

B.2.9 Operadores y puntuadores

operator-or-punctuator:: una de

{	**}**	**[**	**]**	**(**	**)**	**.**	**,**	**:**	**;**
+	**-**	*****	**/**	**%**	**&**	**|**	**^**	**!**	**~**
=	**<**	**>**	**?**	**??**	**::**	**++**	**--**	**&&**	**||**
->	**==**	**!=**	**<=**	**>=**	**+=**	**-=**	***=**	**/=**	**%=**
&=	**|=**	**^=**	**<<**	**<<=**					

right-shift::
> >

right-shift-assignment::
> >=

B.2.10 Directivas de pre-procesamiento

```
pp-directive::
  pp-declaration
  pp-conditional
  pp-line
  pp-diagnostic
  pp-region
  pp-pragma

conditional-symbol::
  identifier
  Cualquier keyword excepto true o false

pp-expression::
  whitespaceopc pp-or-expression whitespaceopc

pp-or-expression::
  pp-and-expression
  pp-or-expression whitespaceopc || whitespaceopc
       pp-and-expression

pp-and-expression::
  pp-equality-expression
  pp-and-expression whitespaceopc && whitespaceopc
       pp-equality-expression

pp-equality-expression::
  pp-unary-expression
  pp-equality-expression whitespaceopc == whitespaceopc
       pp-unary-expression
  pp-equality-expression whitespaceopc != whitespaceopc
       pp-unary-expression

pp-unary-expression::
  pp-primary-expression
  ! whitespaceopc pp-unary-expression

pp-primary-expression::
  true
  false
  conditional-symbol
  ( whitespaceopc pp-expression whitespaceopc )

pp-declaration::
  whitespaceopc # whitespaceopc define whitespace
       conditional-symbol pp-new-line
  whitespaceopc # whitespaceopc undef whitespace conditional-symbol
       pp-new-line

pp-new-line::
  whitespaceopc single-line-commentopc new-line

pp-conditional::
```

pp-if-section pp-elif-sections$_{opc}$ *pp-else-section*$_{opc}$ *pp-endif*

pp-if-section::
whitespace$_{opc}$ **#** *whitespace*$_{opc}$ **if** *whitespace pp-expression*
pp-new-line conditional-section$_{opc}$

pp-elif-sections::
pp-elif-section
pp-elif-sections pp-elif-section

pp-elif-section::
whitespace$_{opc}$ **#** *whitespace*$_{opc}$ **elif** *whitespace pp-expression*
pp-new-line conditional-section$_{opc}$

pp-else-section::
whitespace$_{opc}$ **#** *whitespace*$_{opc}$ **else** *pp-new-line*
conditional-section$_{opc}$

pp-endif::
whitespace$_{opc}$ **#** *whitespace*$_{opc}$ **endif** *pp-new-line*

conditional-section::
input-section
skipped-section

skipped-section::
skipped-section-part
skipped-section skipped-section-part

skipped-section-part::
whitespace$_{opc}$ *skipped-characters*$_{opc}$ *new-line*
pp-directive

skipped-characters::
not-number-sign input-characters$_{opc}$

not-number-sign::
Cualquier *input-character* excepto **#**

pp-line::
whitespace$_{opc}$ **#** *whitespace*$_{opc}$ **line** *whitespace line-indicator*
pp-new-line

line-indicator::
decimal-digits whitespace file-name
decimal-digits
identifier-or-keyword

file-name::
" *file-name-characters* **"**

file-name-characters::
file-name-character
file-name-characters file-name-character

file-name-character::
Cualquier carácter excepto **"** *(***U+0022***)*, y *new-line-character*

```
pp-diagnostic::
   whitespaceopc # whitespaceopc error pp-message
   whitespaceopc # whitespaceopc warning pp-message

pp-message::
   new-line
   whitespace input-charactersopc new-line

pp-region::
   pp-start-region conditional-sectionopc pp-end-region

pp-start-region::
   whitespaceopc # whitespaceopc region pp-message

pp-end-region::
   whitespaceopc # whitespaceopc endregion pp-message

pp-pragma:
   whitespaceopc # whitespaceopc pragma pp-pragma-text

pp-pragma-text:
   new-line
   whitespace input-charactersopc new-line
```

B.3 A.2 Gramática sintáctica

B.3.1 Conceptos básicos

```
compilation-unit:
   extern-alias-directivesopc using-directivesopc
        global-attributesopc namespace-member-declarationsopc

namespace-name:
   namespace-or-type-name

type-name:
   namespace-or-type-name

namespace-or-type-name:
   identifier type-argument-listopc
   qualified-alias-member
   namespace-or-type-name . identifier type-argument-listopc
```

B.3.2 Tipos

```
type:
   value-type
   reference-type
   type-parameter
```

```
value-type:
  struct-type
  enum-type

struct-type:
  type-name
  simple-type
  nullable-type

simple-type:
  numeric-type
  bool

numeric-type:
  integral-type
  floating-point-type
  decimal

integral-type:
  sbyte
  byte
  short
  ushort
  int
  uint
  long
  ulong
  char

floating-point-type:
  float
  double

enum-type:
  type-name

nullable-type:
  non-nullable-value-type ?

non-nullable-value-type:
  enum-type
  type-name
  simple-type

reference-type:
  class-type
  interface-type
  array-type
  delegate-type

class-type:
  type-name
  object
  string

interface-type:
```

```
    type-name

array-type:
    non-array-type rank-specifiers

non-array-type:
    value-type
    class-type
    interface-type
    delegate-type
    type-parameter

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    [ dim-separatorsopc ]

dim-separators:
    ,
    dim-separators ,

delegate-type:
    type-name
```

B.3.3 Variables

```
variable-reference:
    expression
```

B.3.4 Expresiones

```
argument-list:
    argument
    argument-list , argument

argument:
    expression
    ref variable-reference
    out variable-reference

primary-expression:
    array-creation-expression
    primary-no-array-creation-expression

primary-no-array-creation-expression:
    literal
    simple-name
    parenthesized-expression
    member-access
    invocation-expression
```

element-access
this-access
base-access
post-increment-expression
post-decrement-expression
object-creation-expression
delegate-creation-expression
typeof-expression
checked-expression
unchecked-expression
default-value-expression
anonymous-method-expression

simple-name:
identifier type-argument-list$_{opc}$

parenthesized-expression:
(*expression* **)**

member-access:
primary-expression **.** *identifier type-argument-list*$_{opc}$
predefined-type **.** *identifier type-argument-list*$_{opc}$
qualified-alias-member **.** *identifier type-argument-list*$_{opc}$

predefined-type: una de
bool **byte** **char** **decimal** **double** **float**
int **long** **object** **sbyte** **short** **string**
uint **ulong** **ushort**

invocation-expression:
primary-expression **(** *argument-list*$_{opc}$ **)**

element-access:
primary-no-array-creation-expression **[** *expression-list* **]**

expression-list:
expression
expression-list **,** *expression*

this-access:
this

base-access:
base . *identifier type-argument-list*$_{opc}$
base [*expression-list* **]**

post-increment-expression:
primary-expression **++**

post-decrement-expression:
primary-expression **--**

object-creation-expression:
new *type* **(** *argument-list*$_{opc}$ **)**

array-creation-expression:

```
    new non-array-type [ expression-list ] rank-specifiers_opc
        array-initializer_opc
    new array-type array-initializer

delegate-creation-expression:
    new delegate-type ( expression )

typeof-expression:
    typeof ( type )
    typeof ( unbound-type-name )
    typeof ( void )

unbound-type-name:
    identifier generic-dimension-specifier_opc
    identifier :: identifier generic-dimension-specifier_opc
    unbound-type-name . identifier generic-dimension-specifier_opc

generic-dimension-specifier:
    < commas_opc >

commas:
    ,
    commas ,

checked-expression:
    checked ( expression )

unchecked-expression:
    unchecked ( expression )

default-value-expression:
    default ( type )

anonymous-method-expression:
    delegate anonymous-method-signature_opc block

anonymous-method-signature:
    ( anonymous-method-parameter-list_opc )

anonymous-method-parameter-list:
    anonymous-method-parameter
    anonymous-method-parameter-list , anonymous-method-parameter

anonymous-method-parameter:
    parameter-modifier_opc type identifier

unary-expression:
    primary-expression
    + unary-expression
    - unary-expression
    ! unary-expression
    ~ unary-expression
    pre-increment-expression
    pre-decrement-expression
    cast-expression

pre-increment-expression:
```

```
   ++ unary-expression

pre-decrement-expression:
   -- unary-expression

cast-expression:
   ( type ) unary-expression

multiplicative-expression:
   unary-expression
   multiplicative-expression * unary-expression
   multiplicative-expression / unary-expression
   multiplicative-expression % unary-expression

additive-expression:
   multiplicative-expression
   additive-expression + multiplicative-expression
   additive-expression - multiplicative-expression

shift-expression:
   additive-expression
   shift-expression << additive-expression
   shift-expression right-shift additive-expression

relational-expression:
   shift-expression
   relational-expression < shift-expression
   relational-expression > shift-expression
   relational-expression <= shift-expression
   relational-expression >= shift-expression
   relational-expression is type
   relational-expression as type

equality-expression:
   relational-expression
   equality-expression == relational-expression
   equality-expression != relational-expression

and-expression:
   equality-expression
   and-expression & equality-expression

exclusive-or-expression:
   and-expression
   exclusive-or-expression ^ and-expression

inclusive-or-expression:
   exclusive-or-expression
   inclusive-or-expression | exclusive-or-expression

conditional-and-expression:
   inclusive-or-expression
   conditional-and-expression && inclusive-or-expression

conditional-or-expression:
   conditional-and-expression
```

```
    conditional-or-expression || conditional-and-expression

null-coalescing-expression:
    conditional-or-expression
    conditional-or-expression ?? null-coalescing-expression

conditional-expression:
    null-coalescing-expression
    null-coalescing-expression ? expression : expression

assignment:
    unary-expression assignment-operator expression

assignment-operator: una de
    =    +=   -=   *=   /=   %=   &=   |=   ^=   <<=
         right-shift-assignment

expression:
    conditional-expression
    assignment

constant-expression:
    expression

boolean-expression:
    expression
```

B.3.5 Sentencias

```
statement:
    labeled-statement
    declaration-statement
    embedded-statement

embedded-statement:
    block
    empty-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
    try-statement
    checked-statement
    unchecked-statement
    lock-statement
    using-statement
    yield-statement

block:
    { statement-list_opc }

statement-list:
    statement
    statement-list statement
```

```
empty-statement:
  ;

labeled-statement:
  identifier : statement

declaration-statement:
  local-variable-declaration ;
  local-constant-declaration ;

local-variable-declaration:
  type local-variable-declarators

local-variable-declarators:
  local-variable-declarator
  local-variable-declarators , local-variable-declarator

local-variable-declarator:
  identifier
  identifier = local-variable-initializer

local-variable-initializer:
  expression
  array-initializer

local-constant-declaration:
  const type constant-declarators

constant-declarators:
  constant-declarator
  constant-declarators , constant-declarator

constant-declarator:
  identifier = constant-expression

expression-statement:
  statement-expression ;

statement-expression:
  invocation-expression
  object-creation-expression
  assignment
  post-increment-expression
  post-decrement-expression
  pre-increment-expression
  pre-decrement-expression

selection-statement:
  if-statement
  switch-statement

if-statement:
  if ( boolean-expression ) embedded-statement
  if ( boolean-expression ) embedded-statement else
       embedded-statement

switch-statement:
```

```
    switch ( expression ) switch-block

switch-block:
    { switch-sections[opc] }

switch-sections:
    switch-section
    switch-sections switch-section

switch-section:
    switch-labels statement-list

switch-labels:
    switch-label
    switch-labels switch-label

switch-label:
    case constant-expression :
    default :

iteration-statement:
    while-statement
    do-statement
    for-statement
    foreach-statement

while-statement:
    while ( boolean-expression ) embedded-statement

do-statement:
    do embedded-statement while ( boolean-expression ) ;

for-statement:
    for ( for-initializer[opc] ; for-condition[opc] ; for-iterator[opc] )
        embedded-statement

for-initializer:
    local-variable-declaration
    statement-expression-list

for-condition:
    boolean-expression

for-iterator:
    statement-expression-list

statement-expression-list:
    statement-expression
    statement-expression-list , statement-expression

foreach-statement:
    foreach ( type identifier in expression ) embedded-statement

jump-statement:
    break-statement
    continue-statement
    goto-statement
```

```
    return-statement
    throw-statement

break-statement:
    break ;

continue-statement:
    continue ;

goto-statement:
    goto identifier ;
    goto case constant-expression ;
    goto default ;

return-statement:
    return expression_opc ;

throw-statement:
    throw expression_opc ;

try-statement:
    try block catch-clauses
    try block catch-clauses_opc finally-clause

catch-clauses:
    specific-catch-clauses
    specific-catch-clauses_opc general-catch-clause

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses specific-catch-clause

specific-catch-clause:
    catch ( class-type identifier_opc ) block

general-catch-clause:
    catch block

finally-clause:
    finally block

checked-statement:
    checked block

unchecked-statement:
    unchecked block

lock-statement:
    lock ( expression ) embedded-statement

using-statement:
    using ( resource-acquisition ) embedded-statement

resource-acquisition:
    local-variable-declaration
    expression
```

yield-statement:
  yield return *expression* **;**
  yield break ;

namespace-declaration:
  namespace *qualified-identifier namespace-body* **;**$_{opc}$

qualified-identifier:
  identifier
  qualified-identifier **.** *identifier*

namespace-body:
  { *extern-alias-directives*$_{opc}$ *using-directives*$_{opc}$
      namespace-member-declarations$_{opc}$ **}**

extern-alias-directives:
  extern-alias-directive
  extern-alias-directives extern-alias-directive

extern-alias-directive:
  extern alias *identifier* **;**

using-directives:
  using-directive
  using-directives using-directive

using-directive:
  using-alias-directive
  using-namespace-directive

using-alias-directive:
  using *identifier* **=** *namespace-or-type-name* **;**

using-namespace-directive:
  using *namespace-name* **;**

namespace-member-declarations:
  namespace-member-declaration
  namespace-member-declarations namespace-member-declaration

namespace-member-declaration:
  namespace-declaration
  type-declaration

type-declaration:
  class-declaration
  struct-declaration
  interface-declaration
  enum-declaration
  delegate-declaration

qualified-alias-member:
  identifier **::** *identifier type-argument-list*$_{opc}$

B.3.6 Clases

```
class-declaration:
  attributes opc class-modifiers opc partial opc class identifier
       type-parameter-list opc class-base opc
       type-parameter-constraints-clauses opc class-body ; opc

class-modifiers:
  class-modifier
  class-modifiers class-modifier

class-modifier:
  new
  public
  protected
  internal
  private
  abstract
  sealed
  static

class-base:
  : class-type
  : interface-type-list
  : class-type , interface-type-list

interface-type-list:
  interface-type
  interface-type-list , interface-type

class-body:
  { class-member-declarations opc }

class-member-declarations:
  class-member-declaration
  class-member-declarations class-member-declaration

class-member-declaration:
  constant-declaration
  field-declaration
  method-declaration
  property-declaration
  event-declaration
  indexer-declaration
  operator-declaration
  constructor-declaration
  finalizer-declaration
  static-constructor-declaration
  type-declaration

constant-declaration:
  attributes opc constant-modifiers opc const type
       constant-declarators ;
```

```
constant-modifiers:
  constant-modifier
  constant-modifiers constant-modifier

constant-modifier:
  new
  public
  protected
  internal
  private

constant-declarators:
  constant-declarator
  constant-declarators , constant-declarator

constant-declarator:
  identifier = constant-expression

field-declaration:
  attributes_opc field-modifiers_opc type variable-declarators ;

field-modifiers:
  field-modifier
  field-modifiers field-modifier

field-modifier:
  new
  public
  protected
  internal
  private
  static
  readonly
  volatile

variable-declarators:
  variable-declarator
  variable-declarators , variable-declarator

variable-declarator:
  identifier
  identifier = variable-initializer

variable-initializer:
  expression
  array-initializer

method-declaration:
  method-header method-body

method-header:
  attributes_opc method-modifiers_opc return-type member-name
       type-parameter-list_opc ( formal-parameter-list_opc )
       type-parameter-constraints-clauses_opc

method-modifiers:
```

```
    method-modifier
    method-modifiers method-modifier

method-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern

return-type:
    type
    void

member-name:
    identifier
    interface-type . identifier

method-body:
    block
    ;

formal-parameter-list:
    fixed-parameters
    fixed-parameters , parameter-array
    parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters , fixed-parameter

fixed-parameter:
    attributes_opc parameter-modifier_opc type identifier

parameter-modifier:
    ref
    out

parameter-array:
    attributes_opc params array-type identifier

property-declaration:
    attributes_opc property-modifiers_opc type member-name
        { accessor-declarations }

property-modifiers:
    property-modifier
    property-modifiers property-modifier

property-modifier:
```

new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

accessor-declarations:
get-accessor-declaration set-accessor-declaration$_{opc}$
set-accessor-declaration get-accessor-declaration$_{opc}$

get-accessor-declaration:
attributes$_{opc}$ *accessor-modifier*$_{opc}$ **get** *accessor-body*

set-accessor-declaration:
attributes$_{opc}$ *accessor-modifier*$_{opc}$ **set** *accessor-body*

accessor-modifier:
protected
internal
private
protected internal
internal protected

accessor-body:
block
;

event-declaration:
attributes$_{opc}$ *event-modifiers*$_{opc}$ **event** *type*
variable-declarators **;**
attributes$_{opc}$ *event-modifiers*$_{opc}$ **event** *type member-name*
{ *event-accessor-declarations* **}**

event-modifiers:
event-modifier
event-modifiers event-modifier

event-modifier:
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

event-accessor-declarations:
  add-accessor-declaration remove-accessor-declaration
  remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:
  attributes$_{opc}$ **add** *block*

remove-accessor-declaration:
  attributes$_{opc}$ **remove** *block*

indexer-declaration:
  attributes$_{opc}$ *indexer-modifiers*$_{opc}$ *indexer-declarator*
      { *accessor-declarations* **}**

indexer-modifiers:
  indexer-modifier
  indexer-modifiers indexer-modifier

indexer-modifier:
  new
  public
  protected
  internal
  private
  virtual
  sealed
  override
  abstract
  extern

indexer-declarator:
  type **this [** *formal-parameter-list* **]**
  type interface-type **. this [** *formal-parameter-list* **]**

operator-declaration:
  attributes$_{opc}$ *operator-modifiers operator-declarator*
      operator-body

operator-modifiers:
  operator-modifier
  operator-modifiers operator-modifier

operator-modifier:
  public
  static
  extern

operator-declarator:
  unary-operator-declarator
  binary-operator-declarator
  conversion-operator-declarator

unary-operator-declarator:
  type **operator** *overloadable-unary-operator* **(** *type identifier* **)**

overloadable-unary-operator: una de

+ **-** **!** **~** **++** **--** **true** **false**

binary-operator-declarator:
type **operator** *overloadable-binary-operator* **(** *type identifier* **,** *type identifier* **)**

overloadable-binary-operator: una de
+ **-** ***** **/** **%**
& **|** **^**
<< *right-shift*
== **!=** **>** **<** **>=** **<=**

conversion-operator-declarator:
implicit operator *type* **(** *type identifier* **)**
explicit operator *type* **(** *type identifier* **)**

operator-body:
block
;

constructor-declaration:
$attributes_{opc}$ $constructor\text{-}modifiers_{opc}$ *constructor-declarator* *constructor-body*

constructor-modifiers:
constructor-modifier
constructor-modifiers constructor-modifier

constructor-modifier:
public
protected
internal
private
extern

constructor-declarator:
identifier **(** $formal\text{-}parameter\text{-}list_{opc}$ **)** $constructor\text{-}initializer_{opc}$

constructor-initializer:
: base ($argument\text{-}list_{opc}$ **)**
: this ($argument\text{-}list_{opc}$ **)**

constructor-body:
block
;

static-constructor-declaration:
$attributes_{opc}$ *static-constructor-modifiers identifier* **(** **)** *static-constructor-body*

static-constructor-modifiers:
$\textbf{extern}_{opc}$ **static**
static $\textbf{extern}_{opc}$

static-constructor-body:

```
    block
    ;

finalizer-declaration:
    attributes_opc extern_opc ~ identifier ( )       finalizer-body

finalizer-body:
    block
    ;
```

B.3.7 Estructuras

```
struct-declaration:
    attributes_opc struct-modifiers_opc partial_opc struct identifier
        type-parameter-list_opc struct-interfaces_opc
        type-parameter-constraints-clauses_opc struct-body ;_opc

struct-modifiers:
    struct-modifier
    struct-modifiers struct-modifier

struct-modifier:
    new
    public
    protected
    internal
    private

struct-interfaces:
    : interface-type-list

struct-body:
    { struct-member-declarations_opc }

struct-member-declarations:
    struct-member-declaration
    struct-member-declarations struct-member-declaration

struct-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    static-constructor-declaration
    type-declaration
```

B.3.8 Arreglos

array-type:
  non-array-type rank-specifiers

non-array-type:
  value-type
  class-type
  interface-type
  delegate-type
  type-parameter

rank-specifiers:
  rank-specifier
  rank-specifiers rank-specifier

rank-specifier:
  [*dim-separators*$_{opc}$ **]**

dim-separators:
  ,
  dim-separators **,**

array-initializer:
  { *variable-initializer-list*$_{opc}$ **}**
  { *variable-initializer-list* **,** **}**

variable-initializer-list:
  variable-initializer
  variable-initializer-list **,** *variable-initializer*

variable-initializer:
  expression
  array-initializer

B.3.9 Interfaces

interface-declaration:
  attributes$_{opc}$ *interface-modifiers*$_{opc}$ **partial**$_{opc}$ **interface**
      identifier type-parameter-list$_{opc}$ *interface-base*$_{opc}$
      type-parameter-constraints-clauses$_{opc}$ *interface-body* **;**$_{opc}$

interface-modifiers:
  interface-modifier
  interface-modifiers interface-modifier

interface-modifier:
  new
  public
  protected
  internal
  private

```
interface-base:
  : interface-type-list

interface-body:
  { interface-member-declarations_opc }

interface-member-declarations:
  interface-member-declaration
  interface-member-declarations interface-member-declaration

interface-member-declaration:
  interface-method-declaration
  interface-property-declaration
  interface-event-declaration
  interface-indexer-declaration

interface-method-declaration:
  attributes_opc new_opc return-type identifier type-parameter-list_opc
       ( formal-parameter-list_opc )
       type-parameter-constraints-clauses_opc ;

interface-property-declaration:
  attributes_opc new_opc type identifier { interface-accessors }

interface-accessors:
  attributes_opc get ;
  attributes_opc set ;
  attributes_opc get ; attributes_opc set ;
  attributes_opc set ; attributes_opc get ;

interface-event-declaration:
  attributes_opc new_opc event type identifier ;

interface-indexer-declaration:
  attributes_opc new_opc type this [ formal-parameter-list ]
       { interface-accessors }
```

B.3.10 Enumeraciones

```
enum-declaration:
  attributes_opc enum-modifiers_opc enum identifier enum-base_opc
       enum-body ;_opc

enum-base:
  : integral-type

enum-body:
  { enum-member-declarations_opc }
  { enum-member-declarations , }

enum-modifiers:
  enum-modifier
  enum-modifiers enum-modifier

enum-modifier:
```

```
    new
    public
    protected
    internal
    private

enum-member-declarations:
    enum-member-declaration
    enum-member-declarations , enum-member-declaration

enum-member-declaration:
    attributes_opc identifier
    attributes_opc identifier = constant-expression
```

B.3.11 Delegados

```
delegate-declaration:
    attributes_opc delegate-modifiers_opc delegate return-type
        identifier type-parameter-list_opc
        ( formal-parameter-list_opc )
        type-parameter-constraints-clauses_opc ;

delegate-modifiers:
    delegate-modifier
    delegate-modifiers delegate-modifier

delegate-modifier:
    new
    public
    protected
    internal
    private
```

B.3.12 Atributos

```
global-attributes:
    global-attribute-sections

global-attribute-sections:
    global-attribute-section
    global-attribute-sections global-attribute-section

global-attribute-section:
    [ global-attribute-target-specifier attribute-list ]
    [ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier:
    global-attribute-target :

global-attribute-target:
    identifier
    keyword
```

```
attributes:
  attribute-sections

attribute-sections:
  attribute-section
  attribute-sections attribute-section

attribute-section:
  [ attribute-target-specifier_opc attribute-list ]
  [ attribute-target-specifier_opc attribute-list , ]

attribute-target-specifier:
  attribute-target :

attribute-target:
  identifier
  keyword

attribute-list:
  attribute
  attribute-list , attribute

attribute:
  attribute-name attribute-arguments_opc

attribute-name:
  type-name

attribute-arguments:
  ( positional-argument-list_opc )
  ( positional-argument-list , named-argument-list )
  ( named-argument-list )

positional-argument-list:
  positional-argument
  positional-argument-list , positional-argument

positional-argument:
  attribute-argument-expression

named-argument-list:
  named-argument
  named-argument-list , named-argument

named-argument:
  identifier = attribute-argument-expression

attribute-argument-expression:
  expression
```

B.3.13 Genéricos

```
type-parameter-list:
  < type-parameters >
```

```
type-parameters:
  attributes_opc type-parameter
  type-parameters , attributes_opc type-parameter

type-parameter:
  identifier

type-argument-list:
  < type-arguments >

type-arguments:
  type-argument
  type-arguments , type-argument

type-argument:
  type

type-parameter-constraints-clauses:
  type-parameter-constraints-clause
  type-parameter-constraints-clauses
       type-parameter-constraints-clause

type-parameter-constraints-clause:
  where type-parameter : type-parameter-constraints

type-parameter-constraints:
  primary-constraint
  secondary-constraints
  constructor-constraint
  primary-constraint , secondary-constraints
  primary-constraint , constructor-constraint
  secondary-constraints , constructor-constraint
  primary-constraint , secondary-constraints ,
       constructor-constraint

primary-constraint:
  class-type
  class
  struct

secondary-constraints:
  interface-type
  type-parameter
  secondary-constraints , interface-type
  secondary-constraints , type-parameter

constructor-constraint:
  new ( )
```

B.4 Extensión de la gramática para código inseguro

class-modifier:
  ...
  unsafe

struct-modifier:
  ...
  unsafe

interface-modifier:
  ...
  unsafe

delegate-modifier:
  ...
  unsafe

field-modifier:
  ...
  unsafe

method-modifier:
  ...
  unsafe

property-modifier:
  ...
  unsafe

event-modifier:
  ...
  unsafe

indexer-modifier:
  ...
  unsafe

operator-modifier:
  ...
  unsafe

constructor-modifier:
  ...
  unsafe

finalizer-declaration:
  attributes$_{opc}$ **extern**$_{opc}$ **unsafe**$_{opc}$ **~** *identifier* **( )** *finalizer-body*
  attributes$_{opc}$ **unsafe**$_{opc}$ **extern**$_{opc}$ **~** *identifier* **( )** *finalizer-body*

static-constructor-modifiers:
  extern$_{opc}$ **unsafe**$_{opc}$ **static**
  unsafe$_{opc}$ **extern**$_{opc}$ **static**
  extern$_{opc}$ **static unsafe**$_{opc}$
  unsafe$_{opc}$ **static extern**$_{opc}$
  static extern$_{opc}$ **unsafe**$_{opc}$
  static unsafe$_{opc}$ **extern**$_{opc}$

embedded-statement:
  ...
  unsafe-statement

unsafe-statement:
  unsafe *block*

type:
  value-type
  reference-type
  type-parameter
  pointer-type

pointer-type:
  unmanaged-type *****
  void *

unmanaged-type:
  type

primary-no-array-creation-expression:
  ...
  sizeof-expression

primary-no-array-creation-expression:
  ...
  pointer-member-access
  pointer-element-access

unary-expression:
  ...
  pointer-indirection-expression
  addressof-expression

pointer-indirection-expression:
  ***** *unary-expression*

pointer-member-access:
  primary-expression **->** *identifier type-argument-list*$_{opc}$

pointer-element-access:
  primary-no-array-creation-expression **[** *expression* **]**

addressof-expression:
  & *unary-expression*

sizeof-expression:
  sizeof (*unmanaged-type* **)**

embedded-statement:
  ...
  fixed-statement

fixed-statement:
  fixed (*pointer-type fixed-pointer-declarators* **)**
      embedded-statement

```
fixed-pointer-declarators:
  fixed-pointer-declarator
  fixed-pointer-declarators , fixed-pointer-declarator

fixed-pointer-declarator:
  identifier = fixed-pointer-initializer

fixed-pointer-initializer:
  & variable-reference
  expression

local-variable-initializer:
  expression
  array-initializer
  stackalloc-initializer

stackalloc-initializer:
  stackalloc unmanaged-type [ expression ]
```

Apéndice C

Gramática de C# extendida

A continuación se presenta la extensión a la gramática de C# para soportar la programación por chequeo. Aquí se muestra la extensión de la gramática léxica y sintáctica para soportar la programación por chequeo en el lenguaje C# definidas en el Apéndice A basándose en la gramática definida por la ECMA en ECMA-334 3ra edición en su Anexo A, presentada en el Apéndice B de este documento. Las nuevas construcciones están escritas utilizando la misma notación utilizada por la ECMA en el estándar de C#.

C.1 Sumario

Para dar soporte a las nuevas construcciones se han creado doce nuevas palabras reservadas y tres nuevas palabras contextuales.

Palabras reservadas:

```
aspect
pre
post
```

```
handler
returned
back
checkis
check
checker
contract
require
ensure
```

Palabras contextuales:

```
force
noforce
here
```

También ha sido necesario modificar algunas construcciones ya definidas en el estándar:

Construcciones extendidas:

```
keyword
class-member-declaration
struct-member-declaration
interface-member-declaration
jump-statement
method-body
accessor-body
operator-body
constructor-body
```

Construcciones redefinidas:

```
accessor-declarations
using-statement
interfaces-method-declaration
interface-accessors
```

C.2 Gramática léxica

C.2.1 Palabras claves

keyword:: una de
  ...
  aspect
  pre
  post
  handler
  returned
  back
  checkis
  check
  checker
  contract
  require
  ensure
  invariant

C.3 Gramática sintáctica

C.3.1 Propiedades envolventes

accessor-declarations:
  get-accessor-declaration field-autocontained-declarations$_{opc}$
      set-accessor-declaration$_{opc}$
      field-autocontained-declarations$_{opc}$
  set-accessor-declaration field-autocontained-declarations$_{opc}$
      get-accessor-declaration$_{opc}$
      field-autocontained-declarations$_{opc}$
  field-autocontained-declarations get-accessor-declaration
      field-autocontained-declarations$_{opc}$
      set-accessor-declaration$_{opc}$
      field-autocontained-declarations$_{opc}$
  field-autocontained-declarations set-accessor-declaration
      field-autocontained-declarations$_{opc}$
      get-accessor-declaration$_{opc}$
      field-autocontained-declarations$_{opc}$

field-autocontained-declarations:
  field-autocontained-declaration
  field-autocontained-declarations
      field-autocontained-declaration

```
field-autocontained-declaration:
  attributes_opc type variable-declarators ;
```

C.3.2 Costuras

```
class-member-declaration:
  ...
  aspect-declaration
  checker-declaration
  contract-declaration

struct-member-declaration:
  ...
  aspect-declaration
  checker-declaration
  contract-declaration

interface-member-declaration:
  ...
  interface-aspect-declaration
  interface-checker-declaration
  interface-contract-declaration
```

C.3.3 Construcciones genéricas

```
assignments:
  assignment ;
  assignments assignment ;

declaration-assignment-statement:
  assignment ;
  local-variable-declaration ;
  local-constant-declaration ;

declaration-assignment-statements:
  declaration-assignment-statement
  declaration-assignment-statements
       declaration-assignment-statement

jump-statement:
  ...
  back-statement

back-statement:
  back expression_opc ;
```

C.3.4 Aspectos

C.3.4.1 Aspectos formales

```
interface-aspect-declaration:
  attributes_opc new_opc aspect type-parameter-list_opc
       ( formal-parameter-list_opc ) : ( formal-parameter-list_opc )
       aspect-type-indicator_opc
       type-parameter-constraints-clauses_opc ;

aspect-declaration:
  aspect-header aspect-body

aspect-header:
  attributes_opc aspect-modifiers_opc aspect-name
       type-parameter-list_opc ( formal-parameter-list_opc ) :
       ( formal-parameter-list_opc ) aspect-type-indicator_opc
       type-parameter-constraints-clauses_opc

aspect-modifiers:
  aspect-modifier
  aspect-modifiers aspect-modifier

aspect-modifier:
  new
  public
  protected
  internal
  private
  static
  virtual
  sealed
  override
  abstract

aspect-name:
  aspect
  interface-type . aspect

aspect-type-indicator:
  : aspect-type

aspect-type:
  force
  noforce

aspect-body:
  { aspect-descriptor-declarations }
  composition-aspect-aplication
       { aspect-descriptor-declarations_opc }
  ;
```

```
aspect-descriptor-declarations:
  pre-descriptor-declaration assignments[opc]
       aspect-post-handler-descriptor-declarations[opc]
  post-descriptor-declaration assignments[opc]
       aspect-pre-handler-descriptor-declarations[opc]
  handler-descriptor-declaration assignments[opc]
       aspect-pre-post-descriptor-declarations[opc]
  assignments pre-descriptor-declaration assignments[opc]
       aspect-post-handler-descriptor-declarations[opc]
  assignments post-descriptor-declaration assignments[opc]
       aspect-pre-handler-descriptor-declarations[opc]
  assignments handler-descriptor-declaration assignments[opc]
       aspect-pre-post-descriptor-declarations[opc]

pre-descriptor-declaration:
  pre block

post-descriptor-declaration:
  post block

handler-descriptor-declaration:
  handler handler-block

handler-block:
  { handler-descriptor-statement }

handler-descriptor-statement:
  catch-clauses[opc] finally-clause
  catch-clauses

aspect-pre-post-descriptor-declarations:
  pre-descriptor-declaration assignaments[opc]
       post-descriptor-declaration[opc] assignaments[opc]
  post-descriptor-declaration assignaments[opc]
       pre-descriptor-declaration[opc] assignaments[opc]

aspect-pre-handler-descriptor-declarations:
  pre-descriptor-declaration assignaments[opc]
       handler-descriptor-declaration[opc] assignaments[opc]
  handler-descriptor-declaration assignaments[opc]
       pre-descriptor-declaration[opc] assignaments[opc]

aspect-post-handler-descriptor-declarations:
  post-descriptor-declaration assignaments[opc]
       handler-descriptor-declaration[opc] assignaments[opc]
  handler-descriptor-declaration assignaments[opc]
       post-descriptor-declaration[opc] assignaments[opc]
```

C.3.4.2 Aspectos anónimos

```
anonymous-aspect:
  { anonymous-aspect-descriptor-declarations }

anonymous-aspect-descriptor-declarations:
  pre-descriptor-declaration declaration-assignment-statements[opc]
```

```
        aspect-post-handler-descriptor-declarations_opc
    post-descriptor-declaration
        declaration-assignment-statements_opc
        aspect-pre-handler-descriptor-declarations_opc
    handler-descriptor-declaration
        declaration-assignment-statements_opc
        aspect-pre-post-descriptor-declarations_opc
    declaration-assignment-statements pre-descriptor-declaration
        declaration-assignment-statements_opc
        aspect-post-handler-descriptor-declarations_opc
    declaration-assignment-statements post-descriptor-declaration
        declaration-assignment-statements_opc
        aspect-pre-handler-descriptor-declarations_opc
    declaration-assignment-statements
        handler-descriptor-declaration
        declaration-assignment-statements_opc
        aspect-pre-post-descriptor-declarations_opc

anonymous-aspect-pre-post-descriptor-declarations:
    pre-descriptor-declaration declaration-assignment-statements_opc
        post-descriptor-declaration_opc
        declaration-assignment-statements_opc
    post-descriptor-declaration
        declaration-assignment-statements_opc
        pre-descriptor-declaration_opc
        declaration-assignment-statements_opc

anonymous-aspect-pre-handler-descriptor-declarations:
    pre-descriptor-declaration declaration-assignment-statements_opc
        handler-descriptor-declaration_opc
        declaration-assignment-statements_opc
    handler-descriptor-declaration
        declaration-assignment-statements_opc
        pre-descriptor-declaration_opc
        declaration-assignment-statements_opc

anonymous-aspect-post-handler-descriptor-declarations:
    post-descriptor-declaration
        declaration-assignment-statements_opc
        handler-descriptor-declaration_opc
        declaration-assignment-statements_opc
    handler-descriptor-declaration
        declaration-assignment-statements_opc
        post-descriptor-declaration_opc
        declaration-assignment-statements_opc
```

C.3.4.3 Aspectos formales y aspectos anónimos

```
aspect-no-dismembred-aplication:
    aspect aspect-aplication-declarations

aspect-aplication-declarations:
    aspect-aplication-declaration
```

```
    aspect-aplication-declarations , aspect-aplication-declaration

aspect-aplication-declaration:
    aspect-invocation-expression
    anonymous-aspect
    contract-as-aspect

aspect-invocation-expression:
    type .opc aspect-parameters
    primary-expression .opc aspect-parameters
    local-aspect-declaration
    member-aspect-invocation

aspect-parameters:
    type-argument-listopc ( argument-listopc ) : ( argument-listopc )
        aspect-type-indicatoropc

local-aspect-declaration:
    type identifier .opc aspect-parameters =
        local-variable-initializer
    local-variable-initializer .opc aspect-parameters

member-aspect-invocation:
    aspect-parameters
```

C.3.4.4 Composición de aspectos

```
composition-aspect-aplication:
    aspect composition-aspect-aplication-declarations

composition-aspect-aplication-declarations:
    composition-aspect-aplication-declaration
    composition-aspect-aplication-declarations ,
        composition-aspect-aplication-declaration

composition-aspect-aplication-declaration:
    composition-aspect-invocation-expression
    member-aspect-invocation
    here

composition-aspect-invocation-expression:
    type aspect-parameters
    primary-expression aspect-parameters
```

C.3.4.5 Aspectos desmembrados

```
dismembred-pre-aspect-declaration:
    pre dismembred-pre-descriptor-declarations

dismembred-pre-descriptor-declarations:
    block
    dismembred-pre-descriptor-declarations , block

dismembred-post-aspect-declaration:
```

```
    post dismembred-post-descriptor-declarations

dismembred-post-descriptor-declarations:
    block
    dismembred-post-descriptor-declarations , block

dismembred-handler-aspect-declaration:
    handler dismembred-handler-descriptor-declarations

dismembred-handler-descriptor-declarations:
    handler-block
    dismembred-handler-descriptor-declarations , handler-block
```

C.3.4.6 Aplicación de aspectos

```
aspect-aplications:
    aspect-aplication
    aspect-aplications aspect-aplication

aspect-aplication:
    aspect-no-dismembred-aplication
    dismembred-pre-aspect-declaration
    dismembred-post-aspect-declaration
    dismembred-handler-aspect-declaration
```

C.3.4.7 Aplicación de aspectos y contratos

```
checkable-aplications:
    contract-aplications aspect-aplications_opc
    aspect-aplications

checkeable-block:
    checkable-aplications block

method-body:
    ...
    checkeable-block

accessor-body:
    ...
    checkeable-block

operator-body:
    ...
    checkeable-block

constructor-body:
    ...
    checkeable-block

static-constructor-body:
    ...
    checkeable-block
```

```
using-statement:
  using ( resource-acquisition ) checkable-aplications opc
      embedded-statement
  using checkable-aplications embedded-statement
```

C.3.5 Instrucciones de chequeo

```
embedded-checkis-expressions:
  embedded-checkis-expression commas opc
  embedded-checkis-expressions , embedded-checkis-expression
      commas opc

embedded-checkis-expression:
  boolean-expression
  checkis-instruction

embedded-checkis-contract-expression:
  require contract-invocation-expression
  ensure contract-invocation-expression

embedded-check-expressions:
  embedded-check-expression commas opc
  embedded-check-expressions , embedded-check-expression
      commas opc

embedded-check-expression:
  boolean-expression else throw expression
  check-instruction
  invocation-expression

embedded-check-contract-expression:
  require contract-invocation-expression
  ensure contract-invocation-expression

embedded-strict-check-expressions:
  embedded-strict-check-expression commas opc
  embedded-strict-check-expressions ,
      embedded-strict-check-expression commas opc

embedded-strict-check-expression:
  boolean-expression else throw expression
  strict-check-instruction

checkis-instruction:
  checkis ( embedded-checkis-expressions )
  checkis checker-aplication
  checkis embedded-checkis-contract-expression

check-instruction:
  check ( embedded-check-expressions )
  check checker-aplication
  check embedded-check-contract-expression

strict-check-instruction:
```

```
    check ( embedded-strict-check-expressions )
    check checker-aplication
    check embedded-check-contract-expression

expression:
    ...
    checkis-instruction

statement-expression:
    ...
    check-instruction
```

C.3.6 Chequeadores

C.3.6.1 Construcción de chequeadores

```
interface-checker-declaration:
    attributes_opc new_opc checker type-parameter-list_opc
        ( formal-parameter-list_opc )
        type-parameter-constraints-clauses_opc ;

checker-declaration:
    checker-header checker-body

checker-header:
    attributes_opc checker-modifiers_opc checker-name
        type-parameter-list_opc ( formal-parameter-list_opc )
        type-parameter-constraints-clauses_opc

checker-modifiers:
    checker-modifier
    checker-modifiers checker-modifier

checker-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract

checker-name:
    checker
    interface-type . checker

checker-body:
    checker-block
    ;
```

```
checker-block:
  { checker-implementation }

checker-implementation:
  checker-descriptor-declarations
  embedded-strict-check-expressions

checker-descriptor-declarations:
  check-descriptor-declaration checkis-descriptor-declaration
  checkis-descriptor-declaration check-descriptor-declaration

check-descriptor-declaration:
  check block

checkis-descriptor-declaration:
  checkis block

partial-checker-block:
  { partial-checker-implementation }

partial-checker-implementation:
  check-descriptor-declaration
  embedded-strict-check-expressions
```

C.3.6.2 Aplicación de chequeadores

```
checker-aplication:
  type .opc checker-parameters
  primary-expression .opc checker-parameters
  local-checker-declaration
  member-checker-invocation

checker-parameters:
  type-argument-listopc ( argument-listopc )

local-checker-declaration:
  type identifier .opc checker-parameters =
       local-variable-initializer
  local-variable-initializer .opc checker-parameters

member-checker-invocation:
  checker-parameters
```

C.3.7 Contratos

C.3.7.1 Contratos formales

```
interface-contract-declaration:
  attributesopc newopc contract type-parameter-listopc
       ( formal-parameter-listopc ) : ( formal-parameter-listopc )
       type-parameter-constraints-clausesopc ;
```

```
contract-declaration:
  contract-header contract-body

contract-header:
  attributes_opc contract-modifiers_opc contract-name
       type-parameter-list_opc ( formal-parameter-list_opc ) :
       ( formal-parameter-list_opc )
       type-parameter-constraints-clauses_opc

contract-modifiers:
  contract-modifier
  contract-modifiers aspect-modifier

contract-modifier:
  new
  public
  protected
  internal
  private
  static
  virtual
  sealed
  override
  abstract

contract-name:
  contract
  interface-type . contract

contract-body:
  { contract-descriptor-declarations }
  composition-contract-aplication
       { contract-descriptor-declarations_opc }
  ;

contract-descriptor-declarations:
  require-descriptor-declaration assignments_opc
       ensure-descriptor-declaration_opc assignments_opc
  ensure-descriptor-declaration assignments_opc
       require-descriptor-declaration_opc assignments_opc
  assignments require-descriptor-declaration assignments_opc
       ensure-descriptor-declaration_opc assignments_opc
  assignments ensure-descriptor-declaration assignments_opc
       require-descriptor-declaration_opc assignments_opc

require-descriptor-declaration:
  require checker-block

ensure-descriptor-declaration:
  ensure checker-block
```

C.3.7.2 Contratos anónimos

```
anonymous-contract:
```

```
    { anonymous-contract-descriptor-declarations }

anonymous-contract-descriptor-declarations:
    partial-require-descriptor-declaration
        declaration-assignment-statements opc
        partial-ensure-descriptor-declaration opc
        declaration-assignment-statements opc
    partial-ensure-descriptor-declaration
        declaration-assignment-statements opc
        partial-require-descriptor-declaration opc
        declaration-assignment-statements opc
    declaration-assignment-statements
        partial-require-descriptor-declaration
        declaration-assignment-statements opc
        partial-ensure-descriptor-declaration opc
        declaration-assignment-statements opc
    declaration-assignment-statements
        partial-ensure-descriptor-declaration
        declaration-assignment-statements opc
        partial-require-descriptor-declaration opc
        declaration-assignment-statements opc

partial-require-descriptor-declaration:
    require partial-checker-block

partial-ensure-descriptor-declaration:
    ensure partial-checker-block
```

C.3.7.3 Contratos formales y contratos anónimos

```
contract-no-dismembred-aplication:
    contract contract-aplication-declarations

contract-aplication-declarations:
    contract-aplication-declaration
    contract-aplication-declarations ,
        contract-aplication-declaration

contract-aplication-declaration:
    contract-invocation-expression
    anonymous-contract

contract-invocation-expression:
    type .opc contract-parameters
    primary-expression .opc contract-parameters
    local-contract-declaration
    member-contract-invocation

contract-parameters:
    type-argument-list opc ( argument-list opc ) : ( argument-list opc )

local-contract-declaration:
    type identifier .opc contract-parameters =
        local-variable-initializer
```

```
    local-variable-initializer .opc contract-parameters

member-contract-invocation:
    contract-parameters
```

C.3.7.4 Composición de contratos

```
composition-contract-aplication:
    contract composition-contract-aplication-declarations

composition-contract-aplication-declarations:
    composition-contract-aplication-declaration
    composition-contract-aplication-declarations ,
        composition-contract-aplication-declaration

composition-contract-aplication-declaration:
    composition-contract-invocation-expression
    member-contract-invocation
    here

composition-contract-invocation-expression:
    type contract-parameters
    primary-expression contract-parameters
```

C.3.7.5 Contratos desmembrados

```
dismembred-require-contract-declaration:
    require dismembred-require-descriptor-declarations

dismembred-require-descriptor-declarations:
    partial-checker-block
    dismembred-require-descriptor-declarations , checker-block

dismembred-ensure-contract-declaration:
    ensure dismembred-ensure-descriptor-declarations

dismembred-ensure-descriptor-declarations:
    partial-checker-block
    dismembred-ensure-descriptor-declarations , checker-block
```

C.3.7.6 Aplicación de contratos

```
contract-aplications:
    contract-aplication
    contract-aplications contract-aplication

contract-aplication:
    contract-no-dismembred-aplication
    dismembred-require-contract-declaration
    dismembred-ensure-contract-declaration

contract-as-aspect:
    contract contract-invocation-expression
```

C.3.7.7 Aplicación de contratos en interfaces

```
interfaces-method-declaration:
  attributes_opc new_opc return-type identifier type-parameter-list_opc
       ( formal-parameter-list_opc )
       type-parameter-constraints-clauses_opc
       contract-aplications_opc ;

interface-accessors:
  attributes_opc get contract-aplications_opc ;
  attributes_opc set contract-aplications_opc ;
  attributes_opc get contract-aplications_opc ; attributes_opc set
       contract-aplications_opc ;
  attributes_opc set contract-aplications_opc ; attributes_opc get
       contract-aplications_opc ;
```

Apéndice D

Una forma de implementación

A continuación se presenta una posible forma de implementar las nuevas construcciones de la programación por chequeo en C# utilizando los recursos y las construcciones ya provistas por el lenguaje.

D.1 Transformación para el manejo de entradas y salidas

A continuación se presenta las transformaciones necesarias para poder manejar las acciones que ocurren al entrar a un bloque de código (acciones `pre`) y al salir de este (bien sea por salida normal – acciones `post` – o por salida en caso de error – acciones `handler` –)

Para el manejo de entradas y salidas también se definen las acciones preliminares, que son útiles para declarar variables ya que su contexto es accesible por el bloque de código sobre el cual se hace el manejo de entradas y salidas. En las acciones preliminares se pueden crear e inicializar las variables que contendrán las referencias a aspectos y contratos que aplican sobre el bloque de código.

D.1.1 Transformación para métodos que retornan void

Se deben aplicar las siguientes transformaciones para manejar las entradas y salidas:

1. En la primera línea se debe colocar las acciones preliminares

2. Dentro de llaves se debe llamar a las acciones `pre`, que se ejecutan antes de cualquier instrucción de la implementación del método

3. Todo el cuerpo del método se envuelve dentro de un `try`, y dentro de este, toda ocurrencia de la sentencia

   ```
   return;
   ```

   se debe transformar a

   ```
   goto __returned;
   ```

4. Se cierra la llave del `try` y se hace una captura de todas las excepciones bajo el nombre `__e`, se llama dentro de esta sentencia `catch` a las acciones `handler`, que se ejecutan en caso de error; el relanzar la excepción capturada quede de parte de la implementación del `handler`.

5. Antes de cerrar la llave de la sentencia `catch` se escribe la instrucción

   ```
   goto __return;
   ```

6. Se define la etiqueta `__returned` y luego dentro de llaves se llama a las acciones `post`, que se ejecutan siempre que ocurra una salida normal

7. Y por último, se define la etiqueta `__return` y luego se llama a la instrucción `return;`

Importante: Para dar soporte a la instrucción `back;` en los descriptores, `pre`, `post` y `handler` se debe definir la etiqueta `__back` en los aspectos anónimos y desmembrados después de la última instrucción de las acciones `pre`, y de las acciones `post` de la siguiente manera:

```
__back:;
```

Ejemplo:

```
void MiMetodo(parametros)
{
  // ... (1)
    return valor;
  // ... (2)
}
```

Se traduce a:

```
void MiMetodo(parametros)
{

  // Acciones preliminares

  {
    // Llamada a las acciones PRE
    __back:;
  }

  try {
    // ... (1)
      goto __returned;
    // ... (2)

  } catch (Exception __e) {
    // Llamadas a las acciones HANDLER
    goto __return;
  }

  __returned:
    {
      // Llamadas a las acciones POST
      __back:;
    }

  __return:
```

```
        return;
    }
```

D.1.2 Transformación para métodos que retornan valor

Se deben aplicar las siguientes transformaciones para manejar las entradas y salidas:

1. En la primera línea del método se debe declarar la variable `returned` que es de igual tipo de dato que el tipo de retorno del método

2. Luego se debe colocar las acciones preliminares

3. En las siguientes líneas y dentro de llaves se debe llamar a las acciones `pre`, que se ejecutan antes de cualquier instrucción de la implementación del método

4. Todo el cuerpo del método se envuelve dentro de un `try`, y dentro de este, toda ocurrencia de la sentencia (donde *`valor`* es el valor retornado)

   ```
   return valor;
   ```

   se debe transformar a

   ```
   { returned = valor; goto __returned; }
   ```

5. Se cierra la llave del `try` y se hace una captura de todas las excepciones bajo el nombre `__e`, se llama dentro de esta sentencia `catch` a las acciones `handler`, que se ejecutan en caso de error; el relanzar la excepción capturada quede de parte de la implementación del `handler`.

6. Antes de cerrar la llave de la sentencia `catch` se escribe la instrucción

```
goto __return;
```

7. Se define la etiqueta `__returned` y luego dentro de llaves se llama a las acciones `post`, que se ejecutan siempre que ocurra una salida normal

8. Y por último, se define la etiqueta `__return` y luego se retorna el valor contenido en la variable `returned`

Importante: Para dar soporte a la instrucción `back;` en los descriptores, `pre`, `post` y `handler` se debe definir la etiqueta `__back` en los aspectos anónimos y desmembrados después de la última instrucción de las acciones `pre`, y de las acciones `post` de la siguiente manera:

```
__back:;
```

Ejemplo:

```
int MiMetodo(parametros)
{
  // ... (1)
    return valor;
  // ... (2)
}
```

Se traduce a:

```
int MiMetodo(parametros)
{

  // Acciones preliminares

  int returned;
  {
    // Llamada a las acciones PRE
    __back:;
  }

  try {
    // ... (1)
      { returned = valor; goto __returned; }
    // ... (2)
```

```
    } catch (Exception __e) {
      // Llamadas a las acciones HANDLER
      goto __return;
    }

    __returned:
      {
        // Llamadas a las acciones POST
        __back:;
      }

    __return:
      return returned;
}
```

D.1.3 Transformación para bloques de código

El manejo de entradas y salidas sobre un bloque de código es similar a la de los métodos. A cada bloque de código que requiera del manejo de entradas y salidas se le da un identificador único (*ID*).

Se deben aplicar las siguientes transformaciones para manejar las entradas y salidas:

1. Si el bloque de código está contenido en un método que retorna un valor, en la primera línea del bloque de código, de no estar declarada, se debe declarar la variable `returned` que es de igual tipo de dato que el tipo de retorno del método.

2. Luego se debe colocar las acciones preliminares

3. En las siguientes líneas y dentro de llaves se debe llamar a las acciones `pre`, que se ejecutan antes de cualquier instrucción del bloque de código

4. Todo el cuerpo del bloque de código se envuelve dentro de un `try`, y

dentro de este, toda ocurrencia de la sentencia (donde *`valor`* es el valor retornado y *`ID`* es el identificador del bloque de código)

```
return valor;
```

se debe transformar a

```
{ returned = valor; goto __returned_ID; }
```

Y toda ocurrencia de la sentencia

```
return;
```

se debe transformar a

```
goto __returned_ID;
```

5. Se cierra la llave del `try` y se hace una captura de todas las excepciones bajo el nombre `__e`, se llama dentro de esta sentencia `catch` a las acciones `handler`, que se ejecutan en caso de error; el relanzar la excepción capturada quede de parte de la implementación del `handler`.

6. Antes de cerrar la llave de la sentencia `catch` se escribe la instrucción (donde *`ID`* es el identificador del bloque de código)

```
goto __return_ID;
```

7. Se escribe la instrucción (donde *`ID`* es el identificador del bloque de código)

```
goto __end_ID;
```

8. Se define la etiqueta `__returned_ID` (donde *`ID`* es el identificador del bloque de código) y luego dentro de llaves se llama a las acciones `post`, que

se ejecutan siempre que ocurra una salida normal.

9. Se define la etiqueta `__return_`*`ID`* (donde *`ID`* es el identificador del bloque de código) y luego se llama a la sentencia de salida debido a retorno, para ello:

   - En caso de que la instrucción `using` esté contenida dentro de otra, se escribe la instrucción (donde *`IDContenedor`* es el identificador del bloque de código que contiene al bloque de código actual)

     ```
     goto __returned_IDContenedor;
     ```

   - En caso que la instrucción `using` esté contenida directamente por un método con manejo de entrada y salida se escribe la instrucción

     ```
     goto __returned;
     ```

   - En caso que la instrucción `using` esté contenida directamente por un método sin manejo de entrada y salida que retorna `void` se escribe la instrucción

     ```
     return;
     ```

   - En caso que la instrucción `using` esté contenida directamente por un método sin manejo de entrada y salida que retorna un valor se escribe la instrucción

     ```
     return returned;
     ```

10. Se define la etiqueta `__end_`*`ID`* (donde *`ID`* es el identificador del bloque de código) y luego se llama a las acciones `post` (otra vez), que se ejecutan siempre que ocurra una salida normal.

11. De estar definido el bloque de código haciendo uso de la instrucción `using`, si esta posee argumentos, todo lo anterior es colocado como cuerpo de una instrucción `using` con los argumentos de la primera.

Importante: Para dar soporte a la instrucción `back;` en los descriptores, `pre`, `post` y `handler` se debe definir la etiqueta `__back` en los aspectos anónimos y desmembrados después de la última instrucción de las acciones `pre`, y de las acciones `post` de la siguiente manera:

```
__back:;
```

Ejemplo: Manejando las entradas y salidas del método y de la instrucción `using`

```
int MiMetodo(parametros)
{
  // ... (1)

  using (inicializacion)
  {
    // ... (2)
      return valor;
    // ... (3)
  }

  // ... (4)
}
```

Se traduce a:

```
int MiMetodo(parametros)
{
  int returned;

  // Acciones preliminares de MiMetodo

  {
    // Llamada a las acciones PRE del método
    __back:;
  }
```

```
try {

  // ... (1)

  using (inicializacion) // ID = 1
  {

    // Acciones preliminares de la instrucción using

    {
      // Llamada a las acciones PRE de using
      __back:;
    }
    try {
      // ... (2)
        { returned = valor; goto __returned_1; }
      // ... (3)
    } catch (Exception __e) {
      // Llamadas a las acciones HANDLER de using
      goto __return_1;
    }
    goto __end_1;

    __returned_1:
      /* esta es utilizada en el caso de que se haya
       * llamado a la instrucción return dentro del bloque
       * de código, por lo que se debe culminar el retorno
       */
      {
        // Llamadas a las acciones POST de using
        __back:;
      }

    __return_1:
      goto __returned;

    __end_1:
      /* esta es utilizada en el caso de que no se haya
       * llamado a la instrucción return dentro del bloque
       * de código
       */
      {
        // Llamadas a las acciones POST de using
        __back:;
      }

  } // fin using

  // ... (4)
} catch (Exception __e) {
  // Llamadas a las acciones HANDLER del método
  goto __return;
}
```

```
    __returned:
      {
        // Llamadas a las acciones POST del método
        __back:;
      }

    __return:
      return returned;
}
```

D.1.4 Transformaciones para otras unidades funcionales

En el descriptor `set` de propiedades e indizadores se trata de la misma manera que para métodos que retornan `void`; pero, para el descriptor `get` de propiedades e indizadores, y para la redefinición de operadores se trata de la misma manera que métodos que retornan valor.

D.1.5 Acciones handler opcionales

En algunos casos no se requiere ejecutar ninguna instrucción en las acciones handler, por lo que, de no existir ninguna instrucción para las acciones handler se omite el `try` – `catch` necesario para su manejo.

Ejemplo: Una transformación para métodos que retornan `void` sinutilizar las acciones `handler`

```
void MiMetodo(parametros)
{
  // ... (1)
    return valor;
  // ... (2)
}
```

Se traduce a:

```
void MiMetodo(parametros)
```

```
{

  // Acciones preliminares

  {
    // Llamada a las acciones PRE
    back:;
  }

  // ... (1)
    goto __returned;
  // ... (2)

  __returned:
    {
      // Llamadas a las acciones POST
      __back:;
    }

  __return:
    return;
}
```

D.2 Aspectos de inspección

Para implementar los aspectos de inspección se genera un método para cada descriptor, para poder crear el método subyacente es necesario realizar una serie de transformaciones tanto a los parámetros (por ende, también hay que hacer transformaciones a los argumentos en el momento de llamar a los métodos subyacentes), como en la implementación de cada descriptor, de forma tal que sean métodos válidos.

D.2.1 Transformación del encabezado

Los aspectos de inspección se deben transformar en tres métodos que permitan invocar a cada uno de sus descriptores.

Para un aspecto de inspección que posea los encabezados:

```
aspect(tipo1 arg1):(tipo2 arg2)

aspect(tipo1 arg1):(tipo2 arg2):noforce
```

Se generan los siguientes métodos:

Para el descriptor `pre`

```
void aspect_noforce_pre(
                          tipo1 arg1,
                          Separator __separator,
                          Output<tipo2> arg2
                        )
```

Para el descriptor `post`

```
void aspect_noforce_post(
                          tipo1 arg1,
                          Separator __separator,
                          tipo2 arg2
                        )
```

Para el descriptor `handler`

```
void aspect_noforce_handler(
                            Exception __exception,
                            tipo1 arg1,
                            Separator __separator,
                            Output<tipo2> arg2
                          )
```

D.2.1.1 Reglas de transformación

1. En todos los métodos se listan los parámetros del aspecto en el mismo orden en que fueron escritos, incluyendo el separador de parámetros de ingreso y egreso que se convierte en un parámetro más de tipo `Separator` y de nombre `__separator`

2. Para los parámetros de egreso en los métodos generados para los

descriptores `pre` y `handler` se le cambia su tipo de dato al tipo genérico `Output<T>` y como argumento de tipo recibe el tipo original del parámetro

3. Para el método que representa el descriptor `handler` se agrega un parámetro más que debe ser el primero de la lista que recibe el método, este parámetro adicional es de tipo `Exception`, de nombre `__exception`, y es utilizado para indicarle al método la excepción que está dando origen a su ejecución.

D.2.1.2 Parámetros de ingreso con modificador ref

Todos los parámetros de ingreso que poseen un modificador `ref` lo conservan en los respectivos métodos generados.

Ejemplo:

```
aspect(ref tipo arg):()
```

Traduce a:

```
void aspect_noforce_pre(
                          ref tipo arg,
                          Separator __separator
                        )

void aspect_noforce_post(
                          ref tipo arg,
                          Separator __separator
                        )

void aspect_noforce_handler(
                            Exception __exception,
                            ref tipo arg,
                            Separator __separator
                          )
```

D.2.1.3 Parámetros de ingreso con modificador out

Todos los parámetros de ingreso que posean en modificador `out` lo conservan en el método asociado al descriptor `pre` pero en los descriptores `post` y `handler` que es requerido hacer algunas transformaciones:

1. Se cambia el modificador `out` por el modificador `ref`

2. Al parámetro transformado se le antepone otro parámetro de tipo `OutParameter` cuyo nombre es la concatenación de `__out_` seguido del nombre del parámetro que lleva el modificador `out`

Ejemplo:

```
aspect(out tipo arg):()
```

Traduce a:

```
void aspect_noforce_pre(
                          out tipo arg,
                          Separator __separator
                        )

void aspect_noforce_post(
                          OutParameter __out_arg,
                          ref tipo arg,
                          Separator __separator
                        )

void aspect_noforce_handler(
                            Exception __exception,
                            OutParameter __out_arg,
                            ref tipo arg,
                            Separator __separator
                          )
```

D.2.1.4 Parámetros de egreso con modificador ref

Todos los parámetros de egreso que posean en modificador `ref` lo conservan

en el método asociado al descriptor `post` pero en los descriptores `pre` y `handler` que es requerido hacer algunas transformaciones:

1. El parámetro pierde el modificador

2. Al parámetro transformado se le antepone otro parámetro de tipo `RefParameter` cuyo nombre es la concatenación de `__ref_` seguido del nombre del parámetro que lleva el modificador `ref`

Ejemplo:

```
aspect():(ref tipo arg)
```

Traduce a:

```
void aspect_noforce_pre(
                          Separator __separator,
                          RefParameter __ref_arg,
                          Output<tipo> arg
                        )

void aspect_noforce_post(
                          Separator __separator,
                          ref tipo arg
                        )

void aspect_noforce_handler(
                            Exception __exception,
                            Separator __separator,
                            RefParameter __ref_arg,
                            Output<tipo> arg
                          )
```

D.2.1.5 Parámetros de egreso con modificador out

Todos los parámetros de egreso que posean en modificador `out` lo conservan en el método asociado al descriptor `post` pero en los descriptores `pre` y `handler` que es requerido hacer algunas transformaciones:

1. El parámetro pierde el modificador

2. Al parámetro transformado se le antepone otro parámetro de tipo `OutParameter` cuyo nombre es la concatenación de `__out_` seguido del nombre del parámetro que lleva el modificador `out`

Ejemplo:

```
aspect():(out tipo arg)
```

Traduce a:

```
void aspect_noforce_pre(
                          Separator __separator,
                          OutParameter __out_arg,
                          Output<tipo> arg
                        )

void aspect_noforce_post(
                          Separator __separator,
                          out tipo arg
                        )

void aspect_noforce_handler(
                            Exception __exception,
                            Separator __separator,
                            OutParameter __out_arg,
                            Output<tipo> arg
                          )
```

D.2.1.6 Limitaciones

Con estas transformaciones no es posible expresas aspectos de inspección que posean el modificador `params` en los parámetros de ingreso.

D.2.2 Generación de los métodos subyacentes

La implementación de los métodos subyacentes de los descriptores se hace realizando una serie de transformaciones definidas en las Reglas de transformación

general para aspectos de inspección y las Reglas particulares para el manejo de excepciones en handler.

Ejemplo: para el aspecto

```
aspect(int[] arreglo):()
{

  pre {
    for(int i = 0; i < arreglo.Lenght; i++)

      if(arreglo[i] > 7)
        back;
      else {
        // ...
      }
  }

  post {/*...*/}

  handler {
    catch (MiExcepcion e){
      /*...*/
      throw;
    }
}
```

Se generan los siguientes métodos:

```
  void aspect_noforce_pre(int[] arreglo, Separator __separator)
  {
    for(int i = 0; i < arreglo.Lenght; i++)

      if(arreglo[i] > 7)
        return;
      else {
        // ...
      }
  }


  void aspect_noforce_post(  int[] arreglo, Separator __separator)
  {/*...*/}


  void aspect_noforce_handler(
                                Exception __exception,
                                int[] arreglo,
                                Separator __separator
```

```
                                )
{
  if ( __exception is MiExcepcion ) {
    MiExcepcion e = __exception as MiExcepcion;
    /*...*/
    return;
  }
}
```

D.2.2.1 Reglas de transformación general para aspectos de inspección

El contenido de cada descriptor es colocado en su respectivo método subyacente, siguiendo las siguientes transformaciones:

1. Toda aparición de la sentencia `back;` es reemplazada por la instrucción `return;` excepto en el descriptor `handler` cuyo uso es inválido

2. Toda aparición de la sentencia `throw;` en el descriptor `handler` es reemplazada por la instrucción `return;`

3. Toda aparición de la sentencia `return;` es inválida.

4. Toda aparición de un nombre de variable (denominada aquí como `x`) de tipo `Output<T>` debe ser transformada a `x.Value`

D.2.2.2 Reglas particulares para el manejo de excepciones en handler

En el descriptor `handler`, toda aparición de la instrucción `cacth` es reemplazada por una instrucción que compara el tipo del argumento con el tipo indicado por la instrucción catch, esto es:

La instrucción

```
catch (MiExcepcion e) { /* ... */ }
```

se transforma a

```
if (__exception is MiExcepcion) {
  MiExcepcion e = __e as MiExcepcion;
  /* ... */
}
```

La instrucción

```
catch (MiExcepcion) { /* ... */ }
```

se transforma a

```
if (__exception is MiExcepcion) {
  /* ... */
}
```

La instrucción

```
catch { /* ... */ }
```

se transforma a

```
{
  /* ... */
}
```

Si hay más de un `catch` se deben escribir cada `catch` transformado unidos a través de la instrucción `else`.

Ejemplo:

```
catch (MiExcepcion e) { /* ... */ }
catch (OtraExcepcion e) { /* ... */ }
catch { /* ... */ }
```

se transforma a

```
if (__exception is MiExcepcion) {
  MiExcepcion e = __e as MiExcepcion;
  /* ... */
} else if (__exception is OtraExcepcion) {
  OtraExcepcion e = __e as OtraExcepcion;
```

```
  /* ... */
} else {
  /* ... */
}
```

D.2.3 Utilización

A la unidad funcional o bloque de código sobre la cual se aplica el aspecto se le le debe agregar el manejo de entradas y salidas, de aplicarse más de un aspecto se anidan bloques de códigos con manejo de entradas y salidas para cada uno de los aspectos aplicados.

D.2.3.1 Argumentos generados tras la transformación

Para los argumentos que se crearon tras realizar la transformación del encabezado del aspecto para generar los métodos subyacentes, que sean de tipo `Separator`, `OutParameter` y `RefParameter`, al momento de invocar estos métodos se le deben dar el valor de `null` pero indicando explícitamente su tipo.

Ejemplo: Para el aspecto

```
aspect(out tipo arg):()
```

se generó el siguiente método subyacente

```
void aspect_noforce_post(
                           OutParameter __out_arg,
                           ref tipo arg,
                           Separator __separator
                         )
```

la invocación de este método subyacente se debe hacer de la forma

```
aspect_noforce_post(
                     (OutParameter) null,
                     ref arg,
```

```
                    (Separator) null
                )
```

D.2.3.2 Argumentos modificados tras la transformación

Para los argumentos que se le modificaron su tipo de datos tras realizar la transformación del encabezado del aspecto para generar los métodos subyacentes, donde el nuevo tipo de datos es `Output` con tipo genérico el tipo original, al momento de invocar estos métodos se debe crear una nueva instancia del tipo concreto.

Ejemplo: Para el aspecto

```
aspect():(tipo arg)
```

se generó el siguiente método subyacente

```
void aspect_noforce_pre(
                          Separator __separator,
                          Output<tipo> arg
                       )
```

la invocación de este método subyacente se debe hacer de la forma

```
aspect_noforce_pre(
                     (Separator) null,
                     new Output<tipo>()
                  )
```

D.2.3.3 Llamada a las acciones

En cada bloque de acciones se llama al método subyacente generado para el descriptor que le corresponde realizando las transformaciones de argumentos previamente definidas:

- en el bloque de acciones `pre` se llama al método subyacente para el

descriptor `pre`

- en el bloque de acciones `post` se llama al método subyacente para el descriptor `post`
- en el bloque de acciones `handler` se llama al método subyacente para el descriptor `handler`, pero el primer argumento es `__e` que corresponde a la excepción capturada, seguidamente se relanza la excepción haciendo uso de la instrucción `throw;`

Ejemplo:

```
int MiMetodo(char arg1, out double arg2)
  aspect MiAspecto(arg1, out arg2):()
  aspect OtroAspecto():(returned)
{
  int x = (int) arg1;
  return x;
}
```

Traduce a:

```
int MiMetodo(char arg1, out double arg2)
{
  int returned;
  {
    MiAspecto.aspect_noforce_pre(
                                  arg1,
                                  out arg2,
                                  (Separator) null
                                  );
  }

  try {

    {
      {
        OtroAspecto.aspect_noforce_pre(
                                           (Separator) null,
                                           new Output<int>()
                                         );
      }
      try {
           int x = (int) arg1;
           { returned = x; goto __returned_1; }
```

```
      } catch (Exception __e) {
        OtroAspecto.aspect_noforce_handler(
                                          __e,
                                          (Separator) null,
                                          new Output<int>()
                                        );
        throw;
        goto __return_1; // Sentencia innecesaria
      }
      goto __end_1; // Sentencia innecesaria

      __returned_1:
        /* esta es utilizada en el caso de que se haya
         * llamado a la instrucción return dentro del bloque
         * de código, por lo que se debe culminar el retorno
         */
        {
          OtroAspecto.aspect_noforce_post(
                                          (Separator) null,
                                          returned
                                        );
        }

      __return_1:
        goto __returned;

      __end_1: //Sentencias innecesarias
        /* esta es utilizada en el caso de que no se haya
         * llamado a la instrucción return dentro del bloque
         * de código
         */
        {
          OtroAspecto.aspect_noforce_post(
                                          (Separator) null,
                                          returned
                                        );
        }

    }

  } catch (Exception __e) {
    MiAspecto.aspect_noforce_handler(
                                    __e,
                                    arg1,
                                    (OutParameter) null,
                                    ref arg2,
                                    (Separator) null
                                  );
    throw;
    goto __return; // Sentencia innecesaria
  }

  __returned:
```

```
        {
          MiAspecto.aspect_noforce_post(
                                          arg1,
                                          (OutParameter) null,
                                          ref arg2,
                                          (Separator) null
                                        );
        }

      __return:
        return returned;
    }
```

D.3 Aspectos retornantes

Para implementar los aspectos retornantes, al igual que en los aspectos de inspección, se genera un método para cada descriptor, se realiza una serie de transformaciones tanto a los parámetros (por ende, también se hacen transformaciones a los argumentos en el momento de llamar a los métodos subyacentes), como en la implementación de cada descriptor, de forma tal que sean métodos válidos, pero las reglas de transformaciones difieren un poco de las utilizadas en los aspectos de inspección.

D.3.1 Transformación del encabezado

Los aspectos retornantes se deben transformar en tres métodos que permitan invocar a cada uno de sus descriptores.

Para un aspecto retornante que posea el encabezado:

```
aspect(tipo1 arg1):(ref tipo2 arg2):force
```

Se generan los siguientes métodos:

Para el descriptor `pre`

```
bool aspect_force_pre(
                        tipo1 arg1,
                        Separator __separator,
                        ref Output<tipo2> arg2
                      )
```

Para el descriptor `post`

```
void aspect_force_post  (
                          tipo1 arg1,
                          Separator __separator,
                          ref tipo2 arg2
                        )
```

Para el descriptor `handler`

```
bool aspect_force_handler(
                            Exception __exception,
                            tipo1 arg1,
                            Separator __separator,
                            ref Output<tipo2> arg2
                          )
```

Los métodos generados para los descriptores `pre` y `handler` retornan un booleano que indica si se forzó el retorno.

D.3.1.1 Reglas de transformación

1. En todos los métodos se listan los parámetros del aspecto en el mismo orden en que fueron escritos, incluyendo el separador de parámetros de ingreso y egreso que se convierte en un parámetro más de tipo `Separator` y de nombre `__separator`

2. Para los parámetros de egreso en los métodos generados para los descriptores `pre` y `handler` se le cambia su tipo de dato al tipo genérico `Output<T>` y como argumento de tipo recibe el tipo original del parámetro

3. Para el método que representa el descriptor `handler` se agrega un

parámetro más que debe ser el primero de la lista que recibe el método, este parámetro adicional es de tipo `Exception`, de nombre `__exception`, y es utilizado para indicarle al método la excepción que está dando origen a su ejecución.

4. Los parámetros de egreso que posean el modificador `ref` lo conservan en todos los métodos generados.

5. Los parámetros de egreso que posean el modificador `out` reciben una transformación descrita más adelante.

D.3.1.2 Parámetros de ingreso con modificador ref

Todos los parámetros de ingreso que poseen un modificador `ref` lo conservan en los respectivos métodos generados.

Ejemplo:

```
aspect(ref tipo arg):():force
```

Traduce a:

```
bool aspect_force_pre(
                        ref tipo arg,
                        Separator __separator
                      )

void aspect_force_post  (
                          ref tipo arg,
                          Separator __separator
                        )

bool aspect_force_handler(
                            Exception __exception,
                            ref tipo arg,
                            Separator __separator
                          )
```

D.3.1.3 Parámetros de ingreso con modificador out

Todos los parámetros de ingreso que posean en modificador `out` lo conservan en el método asociado al descriptor `pre` pero en los descriptores `post` y `handler` que es requerido hacer algunas transformaciones:

1. Se cambia el modificador `out` por el modificador `ref`

2. Al parámetro transformado se le antepone otro parámetro de tipo `OutParameter` cuyo nombre es la concatenación de `__out_` seguido del nombre del parámetro que lleva el modificador `out`

Ejemplo:

```
aspect(out tipo arg):():force
```

Traduce a:

```
bool aspect_force_pre(
                        out tipo arg,
                        Separator __separator
                      )

void aspect_force_post  (
                          OutParameter __out_arg,
                          ref tipo arg,
                          Separator __separator
                        )

bool aspect_force_handler(
                            Exception __exception,
                            OutParameter __out_arg,
                            ref tipo arg,
                            Separator __separator
                          )
```

D.3.1.4 Parámetros de egreso con modificador ref

Todos los parámetros de egreso que poseen un modificador `ref` lo conservan en los respectivos métodos generados.

Ejemplo:

```
aspect():(ref tipo arg):force
```

Traduce a:

```
bool aspect_force_pre(
                        Separator __separator,
                        ref Output<tipo> arg
                      )

void aspect_force_post(
                         Separator __separator,
                         ref tipo arg
                       )

bool aspect_force_handler(
                            Exception __exception,
                            Separator __separator,
                            ref Output<tipo> arg
                          )
```

D.3.1.5 Parámetros de egreso con modificador out

Todos los parámetros de egreso que posean en modificador `out` lo conservan en el método asociado al descriptor `post` pero en los descriptores `pre` y `handler` que es requerido hacer algunas transformaciones:

1. Se cambia el modificador `out` por el modificador `ref`

2. Al parámetro transformado se le antepone otro parámetro de tipo `OutParameter` cuyo nombre es la concatenación de `__out_` seguido del nombre del parámetro que lleva el modificador `out`

Ejemplo:

```
aspect():(out tipo arg):force
```

Traduce a:

```
bool aspect_force_pre(
                        Separator __separator,
                        OutParameter __out_arg,
                        ref Output<tipo> arg
                      )

void aspect_force_post(
                          Separator __separator,
                          out tipo arg
                        )

bool aspect_force_handler(
                            Exception __exception,
                            Separator __separator,
                            OutParameter __out_arg,
                            ref Output<tipo> arg
                          )
```

D.3.1.6 Limitaciones

Con estas transformaciones no es posible expresas aspectos retornantes que posean el modificador `params` en los parámetros de ingreso.

D.3.2 Generación de los métodos subyacentes

La implementación de los métodos subyacentes de los descriptores se hace realizando una serie de transformaciones definidas en las Reglas de transformación general en aspectos retornantes y las Reglas particulares para el manejo de excepciones en handler (definidas en las transformaciones para aspectos de inspección)

Ejemplo: para el aspecto

```
aspect(int[] arreglo):(ref int salida):force
{

  pre {
    if ( arreglo.Lenght <= 0 ) {
      salida = -1;
      return;
    }

    for(int i = 0; i < arreglo.Lenght; i++)

      if(arreglo[i] > 7)
        back;
      else {
        // ...
      }
  }

  post {
    /*...*/
      back;
    /*...*/
      return;
    /*...*/
  }

  handler {
    catch (MiExcepcion e){
      /*...*/
      throw;
    }
    catch (OtraExcepcion e){
      /*...*/
      salida = -2;
      return;
    }
  }
}
```

Se generan los siguientes métodos:

```
  bool aspect_force_pre(  int[] arreglo, Separator __separator,
                          ref Output<int> salida)
  {
    if ( arreglo.Lenght <= 0 ) {
      salida.Value = -1;
      return true;
    }

    for(int i = 0; i < arreglo.Lenght; i++)
```

```
      if(arreglo[i] > 7)
        retun false;
      else {
        // ...
      }

    return false;
}


void aspect_force_post( int[] arreglo, Separator __separator,
                        ref int salida)
{
  /*...*/
    return;
  /*...*/
    return;
  /*...*/
}


bool aspect_force_handler(
                            Exception __exception,
                            int[] arreglo,
                            Separator __separator,
                            ref Output<int> salida
                          )
{
  if ( __exception is MiExcepcion ) {
    MiExcepcion e = __exception as MiExcepcion;
    /*...*/
    return false;
  } else if ( __exception is OtraExcepcion ) {
    OtraExcepcion e = __exceptios as OtraExcepcion;
    /*...*/
    salida.Value = -2;
    return true;
  }

  return true;
}
```

Importante: El compilador debe verificar que se le haya asignado un valor a todos los parámetros de egreso antes de llamar a la instrucción `return true;` en los descriptores `pre` y `handler`.

D.3.2.1 Reglas de transformación general en aspectos retornantes

El contenido de cada descriptor es colocado en su respectivo método subyacente, siguiendo las siguientes transformaciones:

1. Toda aparición de la sentencia `back;` es reemplazada por la instrucción `return;` en el caso del descriptor `post`; o por la instrucción `return false;` en el caso del descriptor `pre`; en el caso del descriptor `handler` su uso es inválido.

2. Toda aparición de la sentencia `throw;` en el descriptor `handler` es reemplazada por la instrucción `return false;`

3. Toda aparición de la sentencia `return;` en los descriptores `pre` y `handler` es reemplazada por la instrucción `return true;` pero en el descriptor `post` permanece igual.

4. A la implementación del descriptor `pre` se le agrega como última línea la sentencia `return false;`

5. A la implementación del descriptor handler se le agrega como última línea la sentencia `return true;`

6. Toda aparición de un nombre de variable (denominada aquí como `x`) de tipo `Output<T>` debe ser transformada a `x.Value`

D.3.3 Utilización

A la unidad funcional o bloque de código sobre la cual se aplica el aspecto se le debe agregar el manejo de entradas y salidas, de aplicarse más de un aspecto se

anidan bloques de códigos con manejo de entradas y salidas para cada uno de los aspectos aplicados.

D.3.3.1 Argumentos generados tras la transformación

Para los argumentos que se crearon tras realizar la transformación del encabezado del aspecto para generar los métodos subyacentes, que sean de tipo `Separator`, `OutParameter` y `RefParameter`, al momento de invocar estos métodos se le deben dar el valor de `null` pero indicando explícitamente su tipo.

Ejemplo: Para el aspecto

```
aspect(out tipo arg):():force
```

se generó el siguiente método subyacente

```
void aspect_force_post(
                        OutParameter __out_arg,
                        ref tipo arg,
                        Separator __separator
                      )
```

la invocación de este método subyacente se debe hacer de la forma

```
aspect_force_post(
                    (OutParameter) null,
                    ref arg,
                    (Separator) null
                  )
```

D.3.3.2 Argumentos modificados tras la transformación

Para los argumentos que se le modificaron su tipo de datos tras realizar la transformación del encabezado del aspecto para generar los métodos subyacentes, donde el nuevo tipo de datos es `Output` con tipo genérico el tipo original, al momento de invocar estos métodos se debe crear una nueva instancia nombrada del

tipo concreto; el nombre de esta variable es `__chequeo_`*`numero`*, donde *`numero`* es un número único para la variable dentro de la unidad funcional.

Ejemplo: Para el aspecto

```
aspect():(ref tipo arg)
```

se generó el siguiente método subyacente

```
bool aspect_force_pre(
                        Separator __separator,
                        ref Output<tipo> arg
                      )
```

la invocación de este método subyacente se debe hacer de la forma

```
Output<tipo> __chequeo_1 = new Output<tipo>();

aspect_force_pre(
                    (Separator) null,
                    __chequeo_1
                  )
```

D.3.3.3 Llamada a las acciones

En cada bloque de acciones se llama al método subyacente generado para el descriptor que le corresponde realizando las transformaciones de argumentos previamente definidas:

En el bloque de acciones `pre`:

- Se crean las instancias nombradas de los parámetros `Output<T>`
- Se llama al método subyacente para el descriptor `pre`
- Si el método retorna `true` a las variables que estaban en las posiciones de argumentos de tipo `Output` se le asigna `x.Value` (donde `x` es el nombre de la instancia nombrada de tipo `Output<T>` que ocupa su

lugar), luego se llama a la etiqueta `__return` (utilizando la instrucción `goto`) del manejo de entradas y salidas sobre el cual se aplica el aspecto (podría ser `__return` o `__return_`*`codigo`*, donde *`codigo`* es el código del bloque con manejo de entrada y salidas)

En el bloque de acciones `post`:

- Se llama al método subyacente para el descriptor `post`

En el bloque de acciones `handler`:

- Se crean las instancias nombradas de los parámetros `Output`
- Se llama al método subyacente para el descriptor `handler`, pero el primer argumento es `__e` que corresponde a la excepción capturada
- Si el método retorna `false` se relanza la excepción haciendo uso de la instrucción `throw;`
- Después de cerrar la llave del `if` para verificar si el método retorna `flase`, se listan las acciones que se van a ejecutar cuando retorna `true`, a las variables que estaban en las posiciones de argumentos de tipo `Output` se le asigna `x.Value` (donde `x` es el nombre de la instancia nombrada de tipo `Output` que ocupa su lugar); a esto le sigue la llamada a la etiqueta `__return` que ya está contemplada en el manejo de entradas y salidas.

Ejemplo:

```
int MiMetodo(char arg)
  aspect MiAspecto(arg):(ref returned)
{
  return (int)arg;
}
```

Traduce a:

```
int MiMetodo(char arg)
{
  int returned;
  {
    Output<int> __chequeo_1 = new Output<int>();
    if ( MiAspecto.aspect_force_pre(
                                      arg,
                                      (Separator) null,
                                      ref __chequeo_1
                                    )
    ) {
      returned = __chequeo_1.Value;
      goto __return;
    }
  }

  try {

    { returned = (int) arg; goto __returned; }

  } catch (Exception __e) {

    Output<int> __chequeo_1 = new Output<int>();
    if ( MiAspecto.aspect_force_handler(
                                      __e,
                                      arg,
                                      (Separator) null,
                                      ref __chequeo_1
                                    )
    ) {
      throw;
    }
    returned = __chequeo_1.Value;
    goto __return;
  }

  __returned:
    {
      MiAspecto.aspect_force_post(
                                    arg,
                                    (Separator) null,
                                    ref returned
                                  );
    }

  __return:
    return returned;
}
```

D.4 Instrucción check

La implementación de la instrucción `check` se hace aplicándole una serie de transformaciones que generarán un grupo de instrucciones que sustituirán a la instrucción `check` original. Las transformaciones a realizar son:

- Cada prueba representa `if`, donde la condición de la prueba representa la condición del `if`
- Cada instrucción `if` resultante de una prueba se anida dentro de la instrucción `if` resultante de la prueba anterior
- La sentencia a ser ejecutada en la clausula `else` de la prueba es la sentencia a ser ejecutada en la clausula `else` del `if` generado para esa prueba
- La sentencia más anidada, es decir, la que se ejecuta en el caso de que todas las pruebas sean superadas es `;` (ninguna instrucción, la sentencia vacia)
- En caso de que la prueba sea una llamada a un método, no se genera un `if` para esta sino que se coloca la llamada y la siguiente prueba se escribe a continuación de esta (de manera secuencial)

Ejemplo:

```
void Ejemplo(int numero)
{
  // Hacer algo

  check(
    numero >= 0        else throw new NumeroNegativoException(),
    (numero % 2) == 0  else throw new NumeroNoParException(),
    AlgunMetodo(numero),
    numero <= 100      else throw new NumeroMuyGrandeException()
  );
```

```
   // Hacer algo más
}
```

Se transforma a:

```
void Ejemplo(int numero)
{
   // Hacer algo

   if (numero >= 0) {
      if ((numero % 2) == 0) {
         AlgunMetodo(numero),
         if (numero <= 100) {
            ;
         } else
            throw new NumeroMuyGrandeException()
      } else
         throw new NumeroNoParException();
   } else
      throw new NumeroNegativoException();

   // Hacer algo más
}
```

D.5 Instrucción checkis

La implementación de la instrucción `checkis` se creando una variable temporal inicializada por defecto en `false` y aplicándole una serie de transformaciones a las instrucciones del `checkis` original que generarán un grupo de instrucciones que podrán colocar la variable temporal en `true` y donde se utiliza la instrucción `chechis` es colocada la variable temporal (previamente a esta se colocan las instrucciones de la transformación). Las transformaciones a realizar son (estas se colocan antes de la instrucción que utiliza el `checkis`):

- Se crea una variable temporal inicializada en `false`
- Cada prueba representa `if`, donde la condición de la prueba representa

la condición del `if`

- Cada instrucción `if` resultante de una prueba se anida dentro de la instrucción `if` resultante de la prueba anterior
- Cada `if` generado no tiene clausula `else`
- La sentencia más anidada, es decir, la que se ejecuta en el caso de que todas las pruebas sean superadas es asignarle `true` a la variable temporal.
- En caso de que la prueba sea una llamada a un método que no retorne `bool`, no se genera un `if` para esta sino que se coloca la llamada y la siguiente prueba se escribe a continuación de esta (de manera secuencial)
- En caso de la prueba sea una llamada a un método que retorne un `bool`, se considera que esta es una prueba común más, por lo que se genera un `if` para ella.

Ejemplo: (Se asume que `AlgunMetodo` retorna `void`)

```
void Ejemplo(int numero)
{
  bool miVariable;

  // Hacer algo

  miVariable = checkis(
    numero >= 0,
    (numero % 2) == 0,
    AlgunMetodo(numero),
    numero <= 100
  );

  // Hacer algo más
}
```

Se transforma a:

```
void Ejemplo(int numero)
{
  bool miVariable;

  // Hacer algo

  bool __tmp = false;
  if (numero >= 0) {
    if ((numero % 2) == 0) {
      AlgunMetodo(numero),
      if (numero <= 100) {
        __tmp = true;
      }
    }
  }
  miVariable = __tmp;

  // Hacer algo más
}
```

D.5.1 Limitaciones

Esta forma de implementarlo no se comporta bien si la instrucción checkis es utilizada en una serie de pruebas concatenadas con && ó || y no es la primera prueba; el mal comportamiento se debe a que si la prueba anterior falla la siguiente prueba (la instrucción `checkis`) nunca debería ser evaluada pero al ser calculada antes esta ya ha sido evaluada.

D.6 Chequeadores

Para implementar los chequeadores se genera un método para cada descriptor, para poder crear el método subyacente es necesario realizar una serie de transformaciones en la implementación de cada descriptor, de forma tal que sean métodos válidos.

D.6.1 Transformación del encabezado

Los chequeadores se deben transformar en dos métodos que permitan invocar a cada uno de sus descriptores (explícitamente, indicado en la implementación avanzada o implícitamente, indicado en la implementación básica).

Para un chequeador que posea el encabezado:

```
check(tipo arg)
```

Se generan los siguientes métodos:

Para el descriptor `check`

```
void checker_check(
                      tipo arg
                   )
```

Para el descriptor `checkis`

```
bool ckecker_checkis(
                        tipo arg
                     )
```

El método generado para el descriptor `checkis` retorna un booleano que indica si los argumentos pasan la prueba.

D.6.2 Generación de los métodos subyacentes

La generación de los métodos subyacentes depende de cómo se implementó el chequeador: al utilizar la implementación general se deducen los dos métodos a partir de lo allí escrito, pero al emplear la notación avanzada los métodos se implementan según lo escrito para cada uno de sus correspondientes descriptores, pero se le aplicar unas pocas transformaciones.

D.6.2.1 Implementación general

A partir de la implementación general de un chequeador se deduce la implementación de los dos métodos subyacente, el cuerpo de ambos métodos sigue la misma estructura básica:

- Cada prueba representa `if`, donde la condición de la prueba representa la condición del `if`
- Cada instrucción `if` resultante de una prueba se anida dentro de la instrucción `if` resultante de la prueba anterior

Las variaciones requeridas para generar el método subyacente `check` son:

- La sentencia a ser ejecutada en la clausula `else` de la prueba es la sentencia a ser ejecutada en la clausula `else` del `if` generado para esa prueba
- La sentencia más anidada, es decir, la que se ejecuta en el caso de que todas las pruebas sean superadas es la sentencia `return;`

Las variaciones requeridas para generar el método subyacente `checkis` son:

- La clausula `else` de cada `if` resultante de una prueba es la instrucción `return false;`
- La sentencia más anidada, es decir, la que se ejecuta en el caso de que todas las pruebas sean superadas es la sentencia `return true;`

En el caso del checkis otra forma de generar su método subyacente es retornar el resultado de unir todas las pruebas a través del operador &&, lo que sería una concatenación lógica “y” de cada una de las pruebas.

Ejemplo: Para el chequeador

```
static class EsParPositivoChecker
{

  public static checker(int numero)
  {

    numero >= 0      else throw new NumeroNegativoException(),
    (numero % 2)==0  else throw new NumeroNoParException()
  }
}
```

se generan los siguientes métodos:

```
static class EsParPositivoChecker
{

  public static checker_check(int numero)
  {

    if(numero >= 0)
      if((numero % 2)==0)
        return;
      else
        throw new NumeroNoParException();
    else
      throw new NumeroNegativoException();
  }

  public static checker_checkis(int numero)
  {

    if(numero >= 0)
      if((numero % 2)==0)
        return true;
      else
        return false;
    else
      return false;
  }
}
```

La implementación del método subyacente para el descriptor checkis haciendo una concatenación lógica sería:

```
public static checker_checkis(int numero)
{
  return (numero >= 0) && ((numero % 2)==0);
}
```

D.6.2.2 Implementación avanzada para el descriptor check

El cuerpo del método para la implementación del descriptor `check` solo requiere una transformación respecto a como fue implementado: la instrucción `back;` se sustituye por `return;`

Ejemplo: Para la implementación avanzada del descriptor `check`

```
static class EsParPositivoChecker
{

  public static checker(int numero)
  {

    check {
      if (numero >= 0)
        if ( (numero % 2)==0 )
          back;
        else
          throw new NumeroNoParException();

      else
        throw new NumeroNegativoException();
    }

    checkis {
        // ...
    }
  }
}
```

genera el siguiente método:

```
void checker_check(int numero)
{
  if (numero >= 0)
    if ( (numero % 2)==0 )
      return;
    else
      throw new NumeroNoParException();

  else
    throw new NumeroNegativoException();
}
```

D.6.2.3 Implementación avanzada para el descriptor checkis

El cuerpo del método para la implementación del descriptor `checkis` solo requiere dos transformaciones respecto a como fue implementado: la instrucción `back true;` se sustituye por `return true;` y `back false;` se sustituye por `return false;`

Ejemplo: Para la implementación avanzada del descriptor `checkis`

```
static class EsParPositivoChecker
{

  public static checker(int numero)
  {

    check {
        // ...
    }

    checkis {
      if (numero >= 0)
        if ( (numero % 2)==0 )
          back true;
        else
          back false;

      else
        back fasle;
    }
  }
}
```

genera el siguiente método:

```
bool ckecker_checkis(int numero)
{
  if (numero >= 0)
    if ( (numero % 2)==0 )
      return true;
    else
      return false;

  else
    return fasle;
}
```

D.6.3 Utilización

La utilización de los chequeadores se traduce en una llamada al método correspondiente, sin tener que realizar ningún tipo de transformación en los argumentos.

Ejemplo: Para el chequeador

```
static class EsParPositivoChecker
{

  public static checker(int numero)
  {

    numero >= 0      else throw new NumeroNegativoException(),
    (numero % 2)==0  else throw new NumeroNoParException()
  }
}
```

Las llamadas:

```
  check EsParPositivoChecker(2);
  bool resultado = checkis EsParPositivoChecker(2);
```

Se traducen a:

```
  EsParPositivoChecker.checker_check(2);
  bool resultado = EsParPositivoChecker.checker_checkis(2);
```

D.7 Contratos

Para implementar los contratos se genera dos métodos para cada descriptor (un método para cada descriptor en la implementación avanzada), para poder crear el método subyacente es necesario realizar una serie de transformaciones tanto a los parámetros (por ende, también hay que hacer transformaciones a los argumentos en el momento de llamar a los métodos subyacentes), como en la implementación de cada descriptor, de forma tal que sean métodos válidos.

D.7.1 Transformación del encabezado

Los contratos se deben transformar en cuatro métodos que permitan invocar a cada uno de sus descriptores (tanto la versión `check` como la `checkis` para cada descriptor).

Para un contrato que posea el encabezado:

```
contract(tipo1 arg1):(tipo2 arg2)
```

Se generan los siguientes métodos:

Para el descriptor `require` en su versión `check`

```
void contract_require_check(
                      tipo1 arg1,
                      Separator __separator,
                      Output<tipo2> arg2
                    )
```

Para el descriptor `require` en su versión `checkis`

```
bool contract_require_checkis(
                      tipo1 arg1,
                      Separator __separator,
```

```
                        Output<tipo2> arg2
                    )
```

Para el descriptor `ensure` en su versión `check`

```
void contract_ensure_check(
                        tipo1 arg1,
                        Separator __separator,
                        tipo2 arg2
                    )
```

Para el descriptor `ensure` en su versión `checkis`

```
bool contract_ensure_checkis(
                        tipo1 arg1,
                        Separator __separator,
                        tipo2 arg2
                    )
```

Los métodos generados para las versiones `checkis` retorna un booleano que indica si los argumentos pasan la prueba.

D.7.1.1 Reglas de transformación

1. En todos los métodos se listan los parámetros del contrato en el mismo orden en que fueron escritos, incluyendo el separador de parámetros de ingreso y egreso que se convierte en un parámetro más de tipo `Separator` y de nombre `__separator`

2. Para los parámetros de egreso en los métodos generados para el descriptor `require` se le cambia su tipo de dato al tipo genérico `Output<T>` y como argumento de tipo recibe el tipo original del parámetro

D.7.1.2 Limitaciones

Con estas transformaciones no es posible expresas contratos que posean el

modificador `params` en los parámetros de ingreso.

D.7.2 Generación de los métodos subyacentes

Un contrato está compuesto por dos chequeadores, el chequeador `require` y el chequeador `ensure`, por lo que la implementación de los métodos subyacentes de los contratos se hace siguiendo las reglas de generación de los métodos subyacentes de chequeadores aplicado a cada uno, pero haciendo la salvedad que los contratos tienen su propia firma de métodos ya descrita.

Ejemplo: Contrato que rige el calculo de raíces cuadradas (haciendo la salvedad de los errores que puedan ocurrir a causa del redondeo y la precisión finita del tipo de dato usado)

```
static class RaizCuadradaContract
{

  public static contract(double argumento):(double resultado)
  {

    require {
      argumento >= 0 else throw new NumeroNegativoException()
    }

    ensure {
      argumento * argumento == resultado else throw
        new ResultadoErradoException(),
      resultado >= 0 else throw new ResultadoNegativoException()
    }
  }
}
```

La implementación de sus métodos subyacentes es:

```
static class RaizCuadradaContract
{
  void contract_require_check(
                        double argumento,
                        Separator __separator,
                        Output<double> resultado
```

```
                    )
  {
    if (argumento >= 0)
      return;
    else
      throw new NumeroNegativoException();
  }

  bool contract_require_checkis(
                      double argumento,
                      Separator __separator,
                      Output<double> resultado
                    )
  {
    if (argumento >= 0)
      return true;
    else
      return false;
  }

  void contract_ensure_check(
                      double argumento,
                      Separator __separator,
                      double resultado
                    )
  {
    if (argumento * argumento == resultado)
      if (resultado >= 0)
        return;
      else
        throw new ResultadoNegativoException();
    else
      throw new ResultadoErradoException();
  }

  bool contract_ensure_checkis(
                      double argumento,
                      Separator __separator,
                      double resultado
                    )
  {
    if (argumento * argumento == resultado)
      if (resultado >= 0)
        return true;
      else
        return false;
    else
      return false;
  }
}
```

D.7.3 Utilización

A la unidad funcional o bloque de código sobre la cual se aplica el contrato se le debe agregar el manejo de entradas y salidas sin acciones `handler`, de aplicarse más de un contrato se anidan bloques de códigos con manejo de entradas y salidas para cada uno de los contratos aplicados.

D.7.3.1 Argumentos generados tras la transformación

Para los argumentos que se crearon tras realizar la transformación del encabezado del contrato para generar los métodos subyacentes, que sean de tipo `Separator`, al momento de invocar estos métodos se le deben dar el valor de `null` pero indicando explícitamente su tipo.

Ejemplo: Para el contrato

```
contract(tipo arg):()
```

se generó el siguiente método subyacente

```
void contract_require_check(
                    tipo arg,
                    Separator __separator
                  )
```

la invocación de este método subyacente se debe hacer de la forma

```
void contract_require_check(
                    arg,
                    (Separator) null
                  )
```

D.7.3.2 Argumentos modificados tras la transformación

Para los argumentos que se le modificaron su tipo de datos tras realizar la

transformación del encabezado del contrato para generar los métodos subyacentes, donde el nuevo tipo de datos es `Output` con tipo genérico el tipo original, al momento de invocar estos métodos se debe crear una nueva instancia del tipo concreto.

Ejemplo: Para el contrato

```
contract(tipo1 arg1):(tipo2 arg2)
```

se generó el siguiente método subyacente

```
void contract_require_check(
                      tipo1 arg1,
                      Separator __separator,
                      Output<tipo2> arg2
                    )
```

la invocación de este método subyacente se debe hacer de la forma

```
void contract_require_check(
                      arg1,
                      (Separator) null,
                      new Output<tipo2>()
                    )
```

D.7.3.3 Llamada a las acciones

En cada bloque de acciones se llama al método subyacente generado para el descriptor que le corresponde realizando las transformaciones de argumentos previamente definidas:

- en el bloque de acciones `pre` se llama al método subyacente para el descriptor `require` en su versión `check`
- en el bloque de acciones `post` se llama al método subyacente para el descriptor `ensure` en su versión `check`

- no se requiere el bloque de acciones `handler`

Ejemplo:

```
int MiMetodo(char arg1, double arg2)
  contract MiContrato(arg1, arg2):()
  contract OtroContrato(arg1):(returned)
{
  int x = (int) arg1;
  return x;
}
```

Traduce a:

```
int MiMetodo(char arg1, double arg2)
{
  int returned;
  {
    MiContrato.contract_require_check(
                                        arg1,
                                        arg2,
                                        (Separator) null
                                      );
  }

  {
    {
      OtroContrato.contract_require_check(
                                            arg1,
                                            (Separator) null,
                                            new Output<int>()
                                          );
    }
    {
        int x = (int) arg1;
        { returned = x; goto __returned_1; }
    }
    goto __end_1; // Sentencia innecesaria

    __returned_1:
      /* esta es utilizada en el caso de que se haya
       * llamado a la instrucción return dentro del bloque
       * de código, por lo que se debe culminar el retorno
       */
      {
        OtroContrato.contract_ensure_check(
                                             arg1,
                                             (Separator) null,
                                             returned
                                           );
```

```
        }

      __return_1:
        goto __returned;

      __end_1: //Sentencias innecesarias
        /* esta es utilizada en el caso de que no se haya
         * llamado a la instrucción return dentro del bloque
         * de código
         */
        {
          OtroContrato.contract_ensure_check(
                                              arg1,
                                              (Separator) null,
                                              returned
                                            );
        }

    __returned:
      {
        MiContrato.contract_ensure_check (
                                            arg1,
                                            arg2,
                                            (Separator) null
                                          );
      }

    __return:
      return returned;
  }
```

D.8 Aspectos y contratos anónimos o desmembrados

La aplicación de aspectos y contratos de manera anónima o desmembrada se hace de manera similar que la aplicación de aspectos y contratos formales (respectivamente), pero en lugar de hacer la invocación al método subyacente correspondiente se coloca la implementación de este siguiendo las mismas reglas de trasformación ya definidas para cada caso, haciendo la salvedad de:

- Toda aparición de la sentencia `back` se debe transformar en:

  ```
  goto __back;
  ```

- Toda aparición de la sentencia return se debe transformar en una llamada a la etiqueta de retorno del manejo de entradas y salidas actual, que podría ser (según sea el caso):

  `goto __return;` ó `goto __return_codigo;`

  Nota: Si la sentencia `return` viene acompañada de una expresión, previamente se debe asignar a la variable `returned` la expresión.

Ejemplo: (aspecto similar al mostrado en la definición de aspectos anónimos)

```
int MiMetodo(string s)
  pre {
    if (s == null)
      return -1;
      // Error: s es nulo
    else if (s.Length <= 0)
      return -2;
      // Error: s está vacío
  }

  post { return 0; // Resultado correcto }

  handler {
    catch {
      return -3;  // Error: otro tipo de error
    }
  }

{
  if (s == “malo”)
    throw new Exception();

  else
    Console.WriteLine(“   El texto es: {0}”, s);
}
```

Traduce a:

```
int MiMetodo(char arg)
{
  int returned;
  {
    if (s == null)
      {returned = -1; goto __return;}
```

```
        // Error: s es nulo
      else if (s.Length <= 0)
        {returned = -2; goto __return;}
        // Error: s está vacío
    }

    try {

      if (s == "malo")
        throw new Exception();

      else
        Console.WriteLine("   El texto es: {0}", s);

    } catch (Exception __e) {

      {returned = -3; goto __return;}
      // Error: otro tipo de error
    }

    __returned:
      {
        {returned = 0; goto __return;} // Resultado correcto
      }

    __return:
      return returned;
}
```

D.9 Propiedades envolventes

Las propiedades envolventes se deben transformar en propiedades tradicionales y el campo autocontenido se declara fuera de la propiedad con el modificador `private` y queda bajo la responsabilidad del compilador asegurarse que el campo autocontenido solo sea utilizado por su propiedad envolvente, arrojando un error en caso que sea utilizado por cualquier otro miembro.

Ejemplo: La clase

```
class CuentaBancaria {

  public Saldo {
    decimal _saldo;
```

```
      get {
         return _saldo;
      }

      set {
         _saldo = value;
      }
   }

   // ...

}
```

se trasforma a:

```
class CuentaBancaria {

   private decimal _saldo;

   public Saldo {
      get {
         return _saldo;
      }

      set {
         _saldo = value;
      }
   }

   // ...

}
```

y el compilador comprueba que el campo `_saldo` solo sea utilizado por la propiedad `Saldo`, y arroja un error de compilación en caso contrario.

D.10 Clases necesarias

A continuación se muestra la implementación de las clases necesarias para la creación y utilización de los métodos subyacentes que dan soporte a las nuevas construcciones.

D.10.1 Clase OutParameter

```
public class OutParameter
{

   public OutParameter()
   {
   }
}
```

D.10.2 Clase Output<T>

```
public class Output<T>
{
   T _value;

   public T Value
   {
      get { return _value; }
      set { _value = value; }
   }

}
```

D.10.3 Clase RefParameter

```
public class RefParameter
{

   public RefParameter()
   {
   }
}
```

D.10.4 Clase Separator

```
public class Separator
{

   public Separator()
   {
   }
}
```

D.11 Limitaciones

No se ha tenido en cuenta las consideraciones relacionadas con el rendimiento, por lo que la solución presentada no está diseñada para ser la más óptima.

Tampoco se ha presentado los mecanismos para hacer que los contratos se hereden, o se puedan aplicar sobre miembros abstractos o en miembros de interfaces.

Apéndice E

Extensión para el soporte de invariantes

A continuación se presentan una extensión a la programación por chequeo para dar soporte a las invariantes de clases e invariantes de bucles: dos construcciones de la programación por contrato.

E.1 Invariantes de clases

Son un miembro de clase o estructura diseñado para probar el cumplimiento o no de una condición al momento de crear instancias y que deben ser mantenida por todos los miembros de esta.

E.1.1 Descripción

Una invariante de clase (o estructura) es una condición que debe ser satisfecha al finalizar la creación de un objeto y que debe ser mantenida por los miembros funcionales que contenga: métodos, propiedades, aspectos, chequeadores y contratos; de forma tal que la condición sea siempre satisfecha por el objeto al ser

observado externamente.

Las invariantes de clase (o estructura) contiene una condición, que expresa restricciones de consistencia generales que se aplican a cada instancia como un todo; a diferencia de los contratos que caracterizan a las unidades funcionales individuales.

Las invariantes son chequeadas al:

- Terminar la ejecución de un constructor.
- Antes de ejecutar la primera instrucción de una unidad funcional (como un método o una propiedad), antes de ser chequeadas las condiciones `require` del contrato que pudiera tener la unidad funcional.
- Después de ejecutar la última instrucción de una unidad funcional, después de ser chequeadas las condiciones `ensure` del contrato que pudiera tener la unidad funcional.

Restricciones:

- Al finalizar la ejecución del constructor son chequeadas las invariantes heredadas y luego las invariantes definidas en la clase o estructura.
- En una unidad funcional las invariantes (todas las que posea el objeto) son chequeadas si la llamada proviene de un objeto externo al objeto en cuestión, es decir, si la llamada es a una unidad funcional contenida en el mismo objeto (bien porque sean heredadas o estén definidas en la misma clase o estructura en cuestión) no se verifican las invariantes.

E.1.2 Declaración

Para dar soporte a las invariantes en el lenguaje se ha creado un nuevo tipo de miembro, de posible empleo en clases, estructuras e interfaces; así como los campos y los métodos son tipos de miembros utilizables en una clase, estructura o interfaz, las invariantes también lo son y se pueden declarar en los mismos lugares en donde es posible declarar un método.

Sintaxis general:

```
invariant nombre {
   /*...*/
}
```

La declaración de la invariante se hace utilizando la palabra `invariant`, seguido se un identificador; las invariantes no reciben parámetros. La invariante es per se un chequeado, por lo que su implementación se puede siguiendo la implementación básica de chequeadores o se pueden colocar dentro de las llaves los descriptores para la implementación avanzada de chequeadores.

Es posible tener más de una invariante, y cada invariante recibe un nombre que debe ser un identificar único de entre todos los miembros de la clase, estructura o interfaz. Se sugiere el uso del sufijo `Invariant` para nombrar las invariantes.

Ejemplo:

```
class CuentaBancaria {

   decimal _saldo;
   public Saldo {
      get {
         return _saldo;
      }
      set {
         _saldo = value;
```

```
    }
  }

  public invariant SaldoPositivoInvariant {
    _saldo >= 0 else throw new SaldoIncorecctoException();
  }
}
```

E.1.2.1 Regla de accesibilidad de los campos

Todo campo de la clase, estructura o interfaz utilizado en alguna de sus invariantes debe poseer un nivel de visibilidad `private` o `protected`; es decir, una invariante que utilice un campo de la clase, estructura o interfaz que rige con un nivel de visibilidad distinto es inválida.

Ejemplo:

El ejemplo anterior es de la sección anterior es válido ya que `_saldo`, usado en la invariante, es implícitamente `private`.

Ejemplo: variación del ejemplo de `CuentaBancaria` de la sección anterior que lo hace inválido ya que `_saldo` posee un nivel de accesibilidad demasiado permisivo.

```
class CuentaBancaria {

  internal decimal _saldo;
  public Saldo {
    get {
      return _saldo;
    }
    set {
      _saldo = value;
    }
  }

  public invariant SaldoPositivoInvariant {
    _saldo >= 0 else throw new SaldoIncorecctoException();
  }
}
```

Importante: si en la invariante se llama a un método que recibe por argumento a `this`, o se llama a algún método de la clase, estructura o interfaz en cuestión, se asume que ese método puede tener acceso a cualquiera de los campos, por lo que todos los campos de esta deben cumplir esta regla.

E.1.2.2 Modificadores

Una invariante puede recibir los mismos modificadores que un método excepto el modificador `extern`, es decir, los modificadores válidos son: `public`, `protected`, `internal`, `private`, `static`, `virtual`, `sealed`, `new`, `override` y `abstract`, y su utilización se rige bajo las mismas reglas que deben seguirse en el caso de los métodos.

E.1.2.3 Declaración sin implementación

En el caso de no dar implementación a la invariante se coloca punto y coma después de nombrarla.

Sintaxis general:

```
invariant nombre;
```

Esta forma se usa para declarar invariantes en interfaces o aspectos que lleven el modificador `abstract`.

E.1.2.4 Implementación explícita

Para realizar una implementación explícita de una invariante, se omiten los modificadores de este y se antepone al nombre de la invariante el nombre de la interfaz seguido de un punto.

Ejemplo:

```
invariant IMiInterfaz.MiInvariante {/* ... */}
```

E.1.3 Utilización

Una vez que se inicia la ejecución de una unidad funcional (como un método o propiedad) desde un objeto externo la invariante no se chequea hasta que no se alcanza el final de la unidad funcional, por lo que es posible que en algún punto intermedio de esta (o de alguna unidad funcional invocada por esta) la invariante no se satisfaga sin que esto cause algún problema.

Ejemplo:

```
class CuentaBancaria {

  decimal _saldo;
  public Saldo {
    get {
      return _saldo;
    }
    set {
      _saldo = value;
    }
  }

  public invariant SaldoPositivoInvariant {
    _saldo >= 0 else throw new SaldoIncorecctoException();
  }

  public HacerAlgo() {
    decimal temporal = _saldo;
    _saldo = -100; //No hay problema, siempre y cuando no se
                   //alcance el final del método y la invariante
                   //SaldoPositivoInvariant siga sin ser
                   //satisfecha
    // ...
    _saldo = temporal;
  }
}
```

E.1.3.1 Chequear manualmente el cumplimiento de una invariante

Debido a que las invariantes son chequeadores, se pueden invocar como tal, para comprobar el cumplimiento o no de una invariante en un momento dado. Para ello se hace uso de la misma instrucciones empleadas para la invocación chequeadores, pero después de haber escrito la palabra `check` o `checkis` se escribe la palabra `invariant`, luego se indica la invariante en cuestión, para ello se indica el objeto (si es una instancia, o la clase si es estático) seguido de un punto y luego el nombre de la invariante; si la invariante está contenida en la misma clase o estructura (bien sea porque la defina ola herede) basta con colocar solamente el nombre de la invariante.

Ejemplo:

Si se crea una instancia de la clase `CuentaBancaria` definida anteriormente en otra clase de la siguiente manera:

```
CuentaBancaria cuenta = new CuentaBancaria()
```

Para invocar la versión de la invariante que inicia una excepción sería:

```
check invariant cuenta.SaldoPositivoInvariant;
```

Para invocar la versión booleana de la invariante sería:

```
checkis invariant cuenta.SaldoPositivoInvariant;
```

Nota: el poder invocar una invariante desde otra clase (o estructura) diferente a la propia viene restringido al nivel de visibilidad que esta posea.

E.1.3.2 Chequear manualmente el cumplimiento de todas invariantes

Al utilizar invariante se crea implícitamente una invariante de nombre `AllInvariant` que posee el nivel de visibilidad más permisivo utilizado en las otras invariantes. Para chequear manualmente el cumplimiento de todas las invariantes de una clase o estructura (bien sean propias o heredadas) basta con chequear el cumplimiento de la invariante `AllInvariant`

Ejemplo:

Si se crea una instancia de la clase `CuentaBancaria` definida anteriormente en otra clase de la siguiente manera:

```
CuentaBancaria cuenta = new CuentaBancaria()
```

Para invocar la versión que inicia una excepción de todas las invariantes sería:

```
check invariant cuenta.AllInvariant;
```

Para invocar la versión booleana `de todas las invariantes sería:`

```
checkis invariant cuenta.AllInvariant;
```

Nota: el poder invocar la invariante `AllInvariant` desde otra clase (o estructura) diferente a la propia viene restringido al nivel de visibilidad más permisivo que posean las invariantes de la clase o estructura (bien sean propias o heredadas).

E.1.4 Invariantes y herencia

Las invariantes se heredan, y el compromiso de cumplirlas también; por los que las clases derivadas de una clase que implementa una invariante deben

respetarla, la invariante será chequeada al finalizar la ejecución del constructor y al iniciar y finalizar una unidad funcional, aun cuando éstas sean definidas en la clase hija y no en la padre.

E.1.4.1 Invariantes e interfaces

Las interfaces pueden declarar e incluso implementar invariantes, haciendo que las implementaciones de la interfaz estén en la obligación de cumplir dicha invariante.

Ejemplo:

```
interface ICuentaBancaria {

   Saldo {
      get;
   }

   invariant SaldoPositivoInvariant {
      _saldo >= 0 else throw new SaldoIncorecctoException();
   }
}
```

Importante: En una interfaz las invariantes no se consideran automáticamente abstractas, para hacerlas abstractas hay que agregarles el modificador `abstract`.

Importante: Una interfaz que hereda de otra interfaz que posee una invariante abstracta puede implementar dicha invariante, para ello en la implementación debe hacer uso del modificador `override`, si lo que se desea es ocultar la invariante de la interfaz base y definir una nueva se puede emplear el modificador `new`.

E.2 Invariantes de bucles

Es una condición que se debe satisfacer antes de que se ejecute la primera iteración de un bucle, y que ser mantenida en cada iteración; de forma que al terminar la ejecución del bucle ésta siga siendo válida.

E.2.1 Descripción

Una invariante de bucle es una condición que debe ser satisfecha antes de que se ejecute la primera iteración y que cada una de las iteraciones debe mantenerla hasta finalizar el bucle.

Las invariantes de bucle contiene una condición, que expresa restricciones de consistencia generales que se aplican al bucle, y que tiene acceso a las variables de control del mismo.

Las invariantes de bucles son chequeadas al:

- Iniciar cada iteración, antes de que sea ejecutada la primera instrucción.
- Al finalizar el bucle, después de que se ejecute la última instrucción de la última iteración pero antes de haber salido del ámbito de las variables de control (de haber).

Importante: La invariante de bucle es chequeada aún cuando no se realice alguna iteración.

E.2.2 Declaración

La declaración de la invariante se hace usando la palabra `invariant` antes de

la llave de apertura del bucle, seguido de la palabra `invariant` y dentro de llaves va su implementación. La invariante es per se un chequeado, por lo que su implementación se puede siguiendo la implementación básica de chequeadores o se pueden colocar dentro de las llaves el descriptor `check` para la implementación avanzada de chequeadores.

Nota: Al utilizar la implementación avanzada de chequeadores en invariantes de bucles no es requerida la implementación del descriptor `checkis` ya que no se utiliza.

Sintaxis general:

```
while(condición)
  invariant {restricciones}
  instrucción


do
  invariant {restricciones}
  instrucción
while (condición)


for (inicialización; condición; modificación)
  invariant {restricciones}
  instrucción


foreach (tipoElemento elemento in colección)
  invariant {restricciones}
  instrucción
```

Nota: Se puede aplicar más de una invariante de bucle sobre una instrucción iterativa, para ello al cerrar la llave de la invariante anterior se anuncia la siguiente invariante con la palabra `invariant` y se implementa.

E.2.3 Utilización

Las invariantes de bucles se pueden utilizar en cualquiera de las instrucciones iterativas y a diferencia de las invariantes de clases no se heredan ni se pueden desactivar. Su utilización sirve para controlar las variables que pudieran estar cambiando su valor iteración tras iteración.

Ejemplo: Método que calcula el máximo común divisor.

```
int MaximoComunDivisor(int a, int b)
  require { a>0 else throw new ArgumetOutOfRangeException() }
  require { b>0 else throw new ArgumetOutOfRangeException() }
{
  for (int x=a, int y=b ; x!=y ; )
    invariant { x>0 else throw new BucleIncorrectoException() }
    invariant { y>0 else throw new BucleIncorrectoException() }
  {
    if ( x > y )
      x-=y;
    else
      y-=x;
  }
  return x;
}
```

E.3 Extensión a la gramática

A continuación se presentan las construcciones gramaticales necesarias para dar soporte a las invariantes.

E.3.1 Sumario

Para dar soporte a las invariantes se han creado una nueva palabra reservadas

Palabras reservadas:

```
invariant
```

También ha sido necesario modificar algunas construcciones:

Construcciones extendidas:

```
keyword
class-member-declaration
struct-member-declaration
interface-member-declaration
checkis-instruction
check-instruction
strict-check-instruction
```

Construcciones redefinidas:

```
while-statement
do-statement
for-statement
foreach-statement
```

E.3.2 Gramática léxica

```
keyword:: una de
  ...
  invariant
```

E.3.3 Gramática sintáctica

```
class-member-declaration:
  ...
  invariant-declaration

struct-member-declaration:
  ...
  invariant-declaration

interface-member-declaration:
  ...
  interface-invariant-declaration
```

E.3.3.1 Instrucciones de chequeo

```
embedded-checkis-invariant-expression:
  invariant primary-expression

embedded-check-invariant-expression:
```

```
    invariant primary-expression

checkis-instruction:
    ...
    checkis embedded-checkis-invariant-expression

check-instruction:
    ...
    check embedded-check-invariant-expression

strict-check-instruction:
    ...
    check embedded-check-invariant-expression
```

E.3.3.2 Invariantes de clases

```
interface-aspect-declaration:
    attributesopc newopc aspect type-parameter-listopc
        ( formal-parameter-listopc ) : ( formal-parameter-listopc )
        aspect-type-indicatoropc
        type-parameter-constraints-clausesopc ;

invariant-declaration:
    invariant-header invariant-body

invariant-header:
    attributesopc invariant-modifiersopc invariant identifier

invariant-modifiers:
    invariant-modifier
    invariant-modifiers invariant-modifier

invariant-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract

invariant-name:
    identifier
    interface-type . identifier

invariant-body:
    checker-block
    ;

interface-invariant-declaration:
    interface-invariant-header invariant-body
```

```
interface-invariant-header:
  attributes_opc interface-invariant-modifiers_opc invariant
       identifier

interface-invariant-modifiers:
  interface-invariant-modifier
  interface-invariant-modifiers interface-invariant-modifier

interface-invariant-modifier:
  new
  override
  abstract
```

E.3.3.3 Invariante de bucles

```
loop-invariant-declarations:
  loop-invariant-declaration
  loop-invariant-declarations loop-invariant-declaration

loop-invariant-declaration:
  invariant partial-checker-block

while-statement:
  while ( boolean-expression ) loop-invariant-declarations
       embedded-statement

do-statement:
  do loop-invariant-declarations embedded-statement while
       ( boolean-expression ) ;

for-statement:
  for ( for-initializer_opc ; for-condition_opc ; for-iterator_opc )
       loop-invariant-declarations embedded-statement

foreach-statement:
  foreach ( type identifier in expression )
       loop-invariant-declarations embedded-statement
```

E.4 Transformación de las invariantes de clases

Las invariantes se deben transformar en dos métodos que permitan invocar a cada uno de sus descriptores (explícitamente indicado en la implementación avanzada o implícitamente en la implementación básica).

Para una invariante que posea el encabezado:

```
NivelAcceso invariant Nombre
```

Se generan los siguientes métodos:

Para el descriptor `check`

```
NivelAcceso void invariant_Nombre_check()
```

Para el descriptor `checkis`

```
NivelAcceso bool invariant_Nombre_checkis()
```

El método generado para el descriptor `checkis` retorna un booleano que indica si los argumentos pasan la prueba.

E.4.1 Agrupación de las invariantes

Al aplicar al menos una invariante se genera de manera implícita una invariante denominada `AllInvariant` que no es más que la llamada a las otras invariantes, permitiendo así agrupar en una sola todas las invariantes.

E.4.1.1 Agrupación de las invariantes en estructuras

Para toda estructura que posea al menos una invariante (propia o heredada) se genera una invariante especial denominada `AllInvariant` que agrupan la funcionalidad todas las invariantes según su descriptor, y que posee por nivel de acceso el nivel más permisivo de las invariantes implementadas o heredadas.

Para `AllInvariant` se generan los siguientes métodos:

Para el descriptor `check`

```
NivelAcceso void invariant_AllInvariant_check()
```

En este método se hace un llamado al descriptor `check` de cada una de las

invariantes que hereda o implementa la estructura (en ese orden).

Para el descriptor `checkis`

```
NivelAcceso bool invariant_AllInvariant_checkis()
```

En este método se hace un llamado al descriptor `checkis` de cada una de las invariantes que hereda o implementa la estructura (en ese orden) unidas a través del operador &&, lo que sería una concatenación lógica "y" de los resultados , el método retorna el resultado de esa concatenación lógica.

E.4.1.2 Agrupación de las invariantes en clases selladas que heredan de Object

Para toda clase que tenga al menos una invariante (propia o heredada) que posea el modificador `sealed` y herede directamente de `Object` se genera la invariante especial `AllInvariant` de la misma manera que para las estructuras.

E.4.1.3 Agrupación de las invariantes en clases no selladas que heredan de Object

Para toda clase que tenga al menos una invariante (propia o heredada) que no posea el modificador `sealed` y que herede directamente de `Object` se genera una invariante especial denominada `__AcumulatedInvariant` que agrupan la funcionalidad todas las invariantes según su descriptor.

Para `__AcumulatedInvariant` se generan los siguientes métodos:

Para el descriptor `check`

```
protected void __AcumulatedInvariant_check()
```

En este método se hace un llamado al descriptor `check` de cada una de las invariantes que hereda o implementa la clase (en ese orden).

Para el descriptor `checkis`

```
protected bool __AcumulatedInvariant_checkis()
```

En este método se hace un llamado al descriptor `checkis` de cada una de las invariantes que hereda o implementa la clase (en ese orden) unidas a través del operador &&, lo que sería una concatenación lógica "y" de los resultados , el método retorna el resultado de esa concatenación lógica.

También se genera la invariante especial denominada `AllInvariant` que llama a la funcionalidad de `__AcumulatedInvariant` según su descriptor, que posee por nivel de acceso el nivel más permisivo de las invariantes implementadas o heredadas, y que posee el modificador `virtual`.

Para `AllInvariant` se generan los siguientes métodos:

Para el descriptor `check`

```
NivelAcceso virtual void invariant_AllInvariant_check() {
   __AcumulatedInvariant_check();
}
```

Para el descriptor `checkis`

```
NivelAcceso virtual bool invariant_AllInvariant_checkis() {
   return __AcumulatedInvariant_checkis();
}
```

E.4.1.4 Agrupación de invariantes en el resto de las clases

Para toda clase que no esté contemplada en ninguno de los casos anteriores y que defina al menos una invariante (no se toma en cuenta las invariantes heredadas) se genera la invariante especial denominada `__AcumulatedInvariant`

que agrupan la funcionalidad todas las invariantes según su descriptor y que oculta la invariante especial de igual nombre de la clase base.

Para `__AcumulatedInvariant` se generan los siguientes métodos:

Para el descriptor `check`

```
new protected void __AcumulatedInvariant_check() {
  base.__AcumulatedInvariant_check();
  // Llamada al descriptor check de las nuevas invariantes
}
```

En este método se hace un llamado al descriptor `check` de cada una de las invariantes que implementa la clase después de haber invocado al método de igual nombre de la clase base.

Para el descriptor `checkis`

```
new protected bool __AcumulatedInvariant_checkis() {
  return base.__AcumulatedInvariant_checkis() &&
  // Llamada al descriptor checkis de las nuevas invariantes
  ;
}
```

En este método se hace un llamado al descriptor `checkis` de cada una de las invariantes implementa la clase, después de haber invocado al método de igual nombre de la clase base, unidos a través del operador &&, lo que sería una concatenación lógica “y” de los resultados , el método retorna el resultado de esa concatenación lógica.

También se reemplaza la invariante especial denominada `AllInvariant` que llama a la funcionalidad de `__AcumulatedInvariant` según su descriptor, que posee por nivel de acceso el nivel más permisivo de las invariantes implementadas o heredadas, y que posee el modificador `override`.

Para `AllInvariant` se generan los siguientes métodos:

Para el descriptor `check`

```
NivelAcceso override void invariant_AllInvariant_check() {
   __AcumulatedInvariant_check();
}
```

Para el descriptor `checkis`

```
NivelAcceso override bool invariant_AllInvariant_checkis() {
   return __AcumulatedInvariant_checkis();
}
```

E.4.1.5 Orden de permisividad de los niveles de acceso

El orden de permisividad de los modificadores de nivel de acceso es el siguiente, ordenados desde el más permisivo al menos permisivo:

public
internal protected
internal ó ***protected***
private

Ejemplo: si existen dos invariantes en una misma clase, una con nivel de acceso `private` y otra con nivel de acceso `protected` se dice entonces que el nivel de acceso más permisivo entre ellas es `protected`.

Importante: en el caso de existir invariantes con el modificador `internal` y otras con el modificador `protected` se considera que el más permisivo entre ellas es `internal protected`.

E.4.2 Limitaciones

En estas transformaciones no se contempla los problemas relacionados con la concurrencia, tampoco se ha definido la forma en como las invariantes de clase

pasan a formar parte de los contratos de los miembros que afecta.

Apéndice F

Caso de estudio

A continuación se presenta un caso de estudio en el cual se ha seleccionado un software ya existente que ha sido desarrollado por completo sin realizar separación de competencias, en él se ha buscado identificar algunas competencias y calcular el ahorro de código de haberse realizado la separación.

F.1 Pautas para el caso de estudio

- Solo se van a contar líneas de código (no se cuentan ni comentarios ni líneas en blanco)
- No se va a tomar en cuenta el código generado por otras herramientas
- No se va a tomar en cuenta el código no escrito en C# (código SQL, código de ASP.Net y archivos de configuración)
- La cantidad de líneas de código ocupadas por una competencia en todo el código son aproximaciones, basadas en el resultado de calcular cuantas líneas ocupa la competencia (después de haber examinado

varios archivos de muestra) multiplicado por la cantidad de ocurrencias identificadas de esa competencia.

F.2 Sistema seleccionado

El sistema seleccionado para el estudio es SGEM (Sistema de Generación y Envío de Mensajes), este software es utilizado para generar mensajes (bajo demanda o por planificación) y despacharlos por diferentes canales (tales como e-mail, SMS, etc.); este sistema esta escrito en C# y su interfaz está hecha en ASP.Net.

El software objeto de estudio posee 48.786 líneas de código escritas en C# (y no generadas por herramientas) distribuidas en 400 archivos de código fuente.

F.3 Resultados

Se han seleccionado dos competencias para realizar el estudio:

- Competencia de escritura en el logger las entradas que corresponden al nivel debug, en las que se registra que se está ingresando y/o que se se está abandonado un método.
- Competencia de tratamiento de errores de base de datos.

Estas dos competencias se han seleccionado debido a que son las más significativas de entre todas las identificadas en el sistema.

Los resultados se presentan en dos partes, la primera, en la que se estudia la implementación del acceso a la base de datos ; y la segunda en la que se estudia el sistema completo.

F.3.1 Acceso a base de datos

En SGEM la implementación de un método de acceso a base de datos se hace de la siguiente manera: en el método se llama a otro método generado por una herramienta que abstrae las llamadas a los procedimiento almacenados en la base de datos. Por lo que un método típico de acceso a base de datos en SGEM sería como el siguiente:

```
public DataTable ConsultarMensaje(long codigoMensaje)
{
   return consultarMensajeTableAdapter.
     ConsultarMensaje(codigoMensaje);
}
```

A este código se le agregaron el control de errores de acceso a la base de datos:

```
public DataTable ConsultarMensaje(long codigoMensaje)
{
   try
   {
      return consultarMensajeTableAdapter.
        ConsultarMensaje(codigoMensaje);
   }
   catch (SqlException e)
   {
      if (Logger.EstaErrorHabilitado)
        Logger.Error(e, _codigoError, (Ilogable)null);

      throw new SGEMFalloAccesoDatosExcepcion
        (this, _codigoError, _mensajeError, e);
   }
}
```

Y por último le agregaron las llamadas al logger para registrar el ingreso y abandono del método (así esta en SGEM):

```
public DataTable ConsultarMensaje(long codigoMensaje)
{
   DataTable respuesta;

   #region Instrumentacion
```

```
        if (Logger.EstaDebugHabilitado)
          Logger.Debug(Eventos.EntrandoMetodo,
            string.Format("Entrando al metodo {0}",
                          MethodInfo.GetCurrentMethod().Name), null);


        #endregion

        try
        {
          respuesta = consultarMensajeTableAdapter.
            ConsultarMensaje(codigoMensaje);
        }
        catch (SqlException e)
        {
          if (Logger.EstaErrorHabilitado)
            Logger.Error(e, _codigoError, (Ilogable)null);

          throw new SGEMFalloAccesoDatosExcepcion
            (this, _codigoError, _mensajeError, e);
        }

        #region Instrumentacion

        if (Logger.EstaDebugHabilitado)
          Logger.Debug(Eventos.SaliendoMetodo,
            string.Format("Saliendo del metodo {0}",
                          MethodInfo.GetCurrentMethod().Name), null);


        #endregion

        return respuesta;
      }
```

El haber agregado el control de errores y las llamas al logger hizo que un método de dos líneas de código se transformara en un método de 26 líneas de código (la funcionalidad del método representa solo el 7,69 % de su código).

Este problema se podría haber resuelto creando dos aspectos: uno para el manejo de los errores de base de datos y otro para el logger:

```
      static class ManejoErrorBDAspect
      {

        static aspect(object obj, int codigoError, string mensaje):()
        {
```

```
      handler {

        catch (SqlException e)
        {
          if (Logger.EstaErrorHabilitado)
            Logger.Error(e, codigoError, (Ilogable)null);

          throw new SGEMFalloAccesoDatosExcepcion
            (obj, codigoError, mensajeError, e);
        }

      }

    } // fin aspect
  }


  static class RegistoLoggerAspect {
    static aspect():() {

      pre {

        if (Logger.EstaDebugHabilitado)
          Logger.Debug(Eventos.EntrandoMetodo,
            string.Format("Entrando al metodo {0}",
                      ObtenerMetodoInvocante() .Name), null);
      }

      post {

        if (Logger.EstaDebugHabilitado)
          Logger.Debug(Eventos.SaliendoMetodo,
            string.Format("Saliendo del metodo {0}",
                      ObtenerMetodoInvocante() .Name), null);
      }

    } // fin aspect

    private static StackFrame ObtenerMetodoInvocante()
    {

      StackTrace stackTrace = new StackTrace();
      return stackTrace.GetFrame( 2 );  // Entrada en la pila de
             // llamada que contiene la información del invocante

    }

  }
```

Y en el método haber aplicado los dos aspectos:

```
public DataTable ConsultarMensaje(long codigoMensaje)
   aspect RegistoLoggerAspect():()
   aspect ManejoErrorBDAspect(this,_codigoError,_mensajeError):()
{
   return consultarMensajeTableAdapter.
      ConsultarMensaje(codigoMensaje);
}
```

Hacerlo de esta manera, aparte de hacer más legible al método, facilita el mantenimiento del sistema; por ejemplo, si se identifica que es necesario capturar una excepción adicional en el manejo de errores de base de datos solo se tendría que hacer el cambio en un único lugar (y no en los 102 lugares donde se hace actualmente el manejo de errores de base de datos).

El acceso a base de datos del sistema está construido en 18 archivos con un total de 5.847 líneas de códigos conformadas de la siguiente manera:

	Apariciones	*Líneas de código*	*%*
Registro en el logger	*132*	*1.330*	*22,75 %*
Manejo de error de base de datos	*102*	*1.020*	*17,44 %*
Resto de la funcionalidad	*N/A*	*3.497*	*59,81 %*
	Total	5.847	100,00 %

Tabla F.1: *Distribución del código de acceso a base de datos*
Fuente: el autor

De haberse realizado la separación de competencias de la 5.847 líneas de código solo serían necesarias 63,81 % de las líneas originales (3.731, la funcionalidad más la aplicación de cada uno de los aspectos – sin contar lo que ocupan los aspectos –).

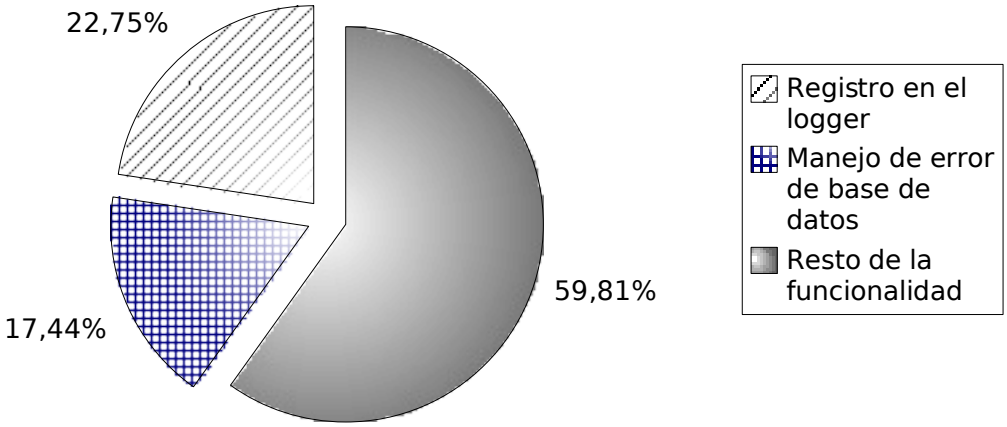


Figura F.1: *Distribución del código de acceso a base de datos*
Fuente: el autor

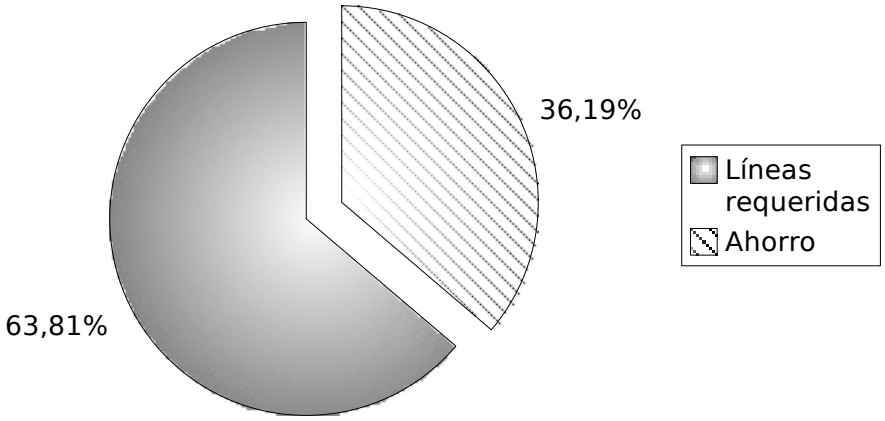


Figura F.2: *Ahorro de código utilizando separación de competencias en el acceso a base de datos*
Fuente: el autor

F.3.2 El sistema

En SGEM el registro en el logger para ingreso y/o abandonado un método se hace en todas las capas, a continuación se muestra lo que ocupa esta competencia frente al resto de las líneas de código.

SGEM está dividido en las siguientes partes:

	Archivos	*Líneas de código*	*%*
Capa web	*37*	*15.935*	*32,66 %*
Capa transaccional	*251*	*18.370*	*37,65 %*
Capa de acceso a datos	*44*	*6.597*	*13,52 %*
Demonios	*30*	*1.991*	*4,08 %*
Otros componentes	*38*	*5.893*	*12,08 %*
Total	400	48.786	100,00 %

Tabla F.2: *Partes de SGEM*
Fuente: el autor

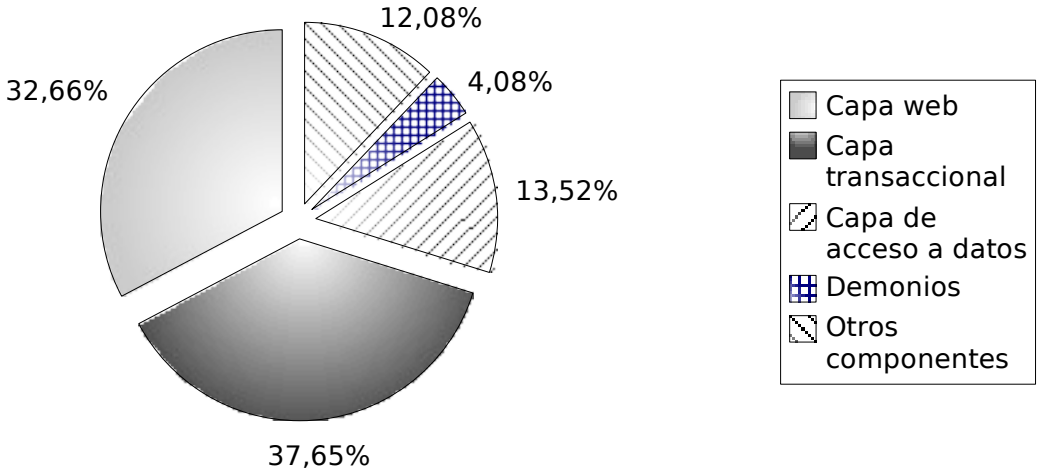


Figura F.3: *Partes de SGEM*
Fuente: el autor

F.3.2.1 Capa web

La capa web del sistema está construida en 37 archivos con un total de 15.935 líneas de códigos conformadas de la siguiente manera:

	Apariciones	*Líneas de código*	*%*
Registro en el logger	*456*	*4.560*	*28,62 %*
Manejo de error de base de datos	*0*	*0*	*0 %*
Resto de la funcionalidad	*N/A*	*11.375*	*71,38 %*
	Total	15.935	100,00 %

Tabla F.3: *Distribución del código de la capa web*
Fuente: el autor

De haberse realizado la separación de competencias de la 15.935 líneas de código solo serían necesarias 74,25 % de las líneas originales (11.831, la funcionalidad más la aplicación del aspecto de registro en logger – sin contar lo que este ocupa –).

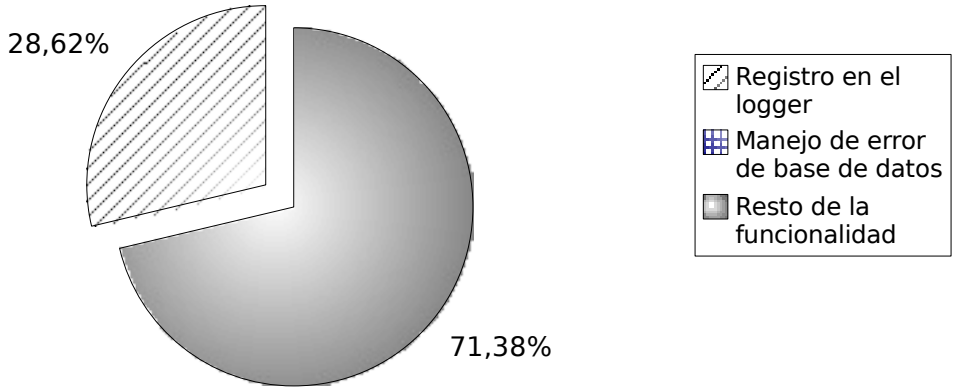


Figura F.4: *Distribución del código de la capa web*
Fuente: el autor

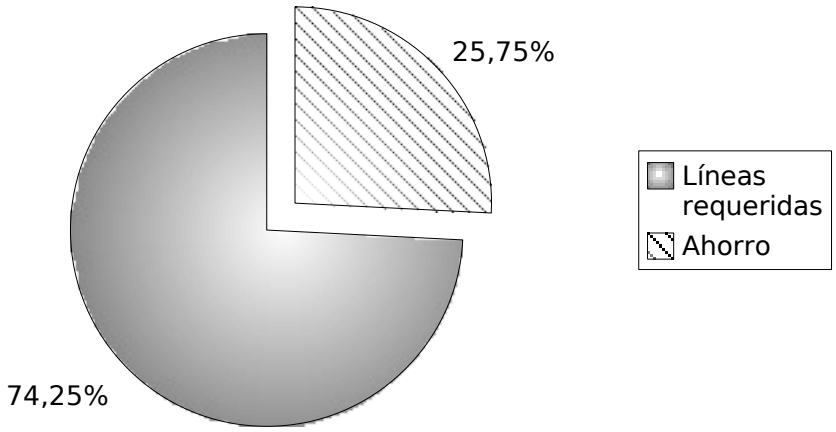


Figura F.5: *Ahorro de código utilizando separación de competencias en la capa web*
Fuente: el autor

F.3.2.2 Capa transaccional

La capa transaccional del sistema está construida en 251 archivos con un total de 18.370 líneas de códigos conformadas de la siguiente manera:

	Apariciones	*Líneas de código*	*%*
Registro en el logger	*247*	*2.495*	*13,58 %*
Manejo de error de base de datos	*0*	*0*	*0 %*
Resto de la funcionalidad	*N/A*	*15.875*	*86,42 %*
	Total	18.370	100,00 %

Tabla F.4: *Distribución del código de la capa transaccional*
Fuente: el autor

De haberse realizado la separación de competencias de la 18.370 líneas de código solo serían necesarias 87,76 % de las líneas originales (16.122, la funcionalidad más la aplicación del aspecto de registro en logger – sin contar lo que este ocupa –).

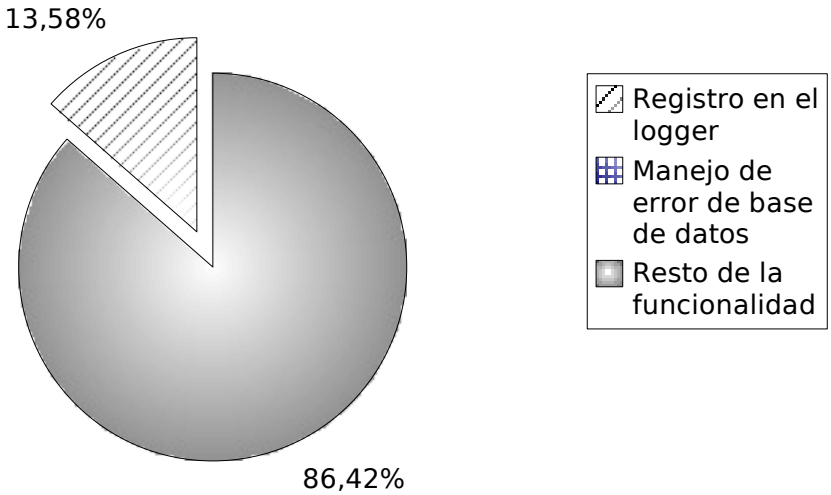


Figura F.6: *Distribución del código de la capa transaccional*
Fuente: el autor

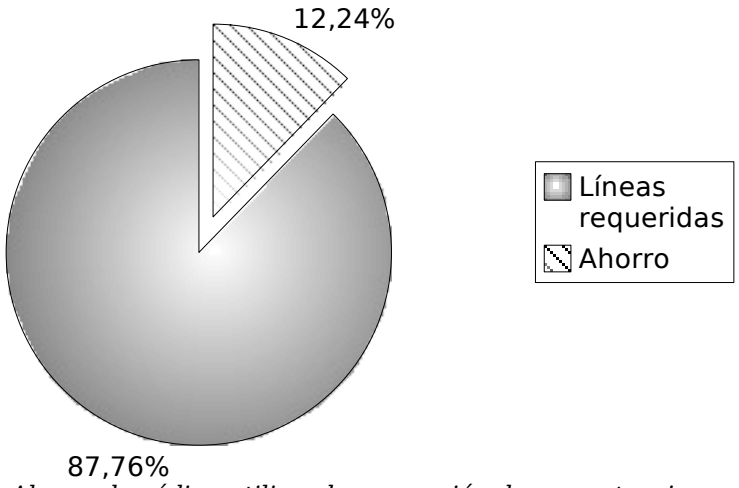


Figura F.7: *Ahorro de código utilizando separación de competencias en la capa transaccional*
Fuente: el autor

F.3.2.3 Capa de acceso a datos

La capa transaccional del sistema está construida en 44 archivos con un total de 6.597 líneas de códigos conformadas de la siguiente manera:

	Apariciones	*Líneas de código*	*%*
Registro en el logger	*133*	*1.340*	*15,46 %*
Manejo de error de base de datos	*102*	*1.020*	*20,31 %*
Resto de la funcionalidad	*N/A*	*4.237*	*64,23 %*
	Total	6.597	100,00 %

Tabla F.5: *Distribución del código de la capa de acceso a datos*
Fuente: el autor

De haberse realizado la separación de competencias de la 6.597 líneas de código solo serían necesarias 67,79 % de las líneas originales (4.472, la funcionalidad más la aplicación de cada uno de los aspectos – sin contar lo que ocupan los aspectos –).

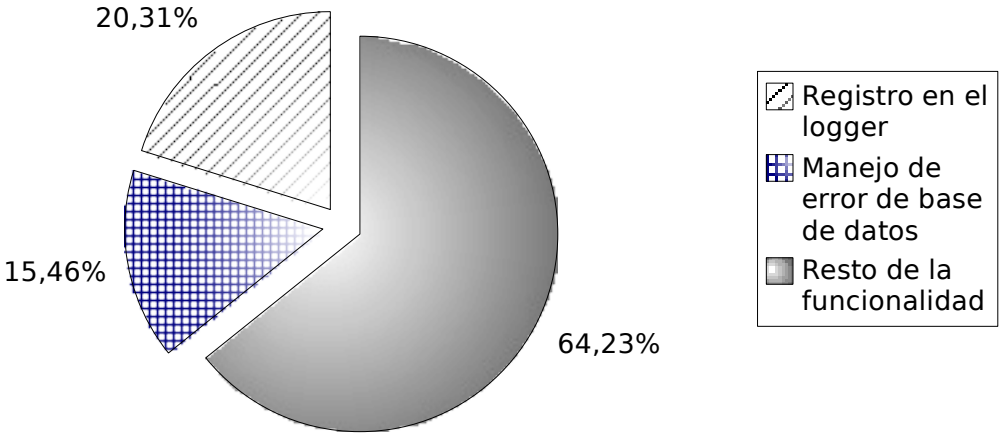


Figura F.8: *Distribución del código de la capa de acceso a dato*
Fuente: el autor

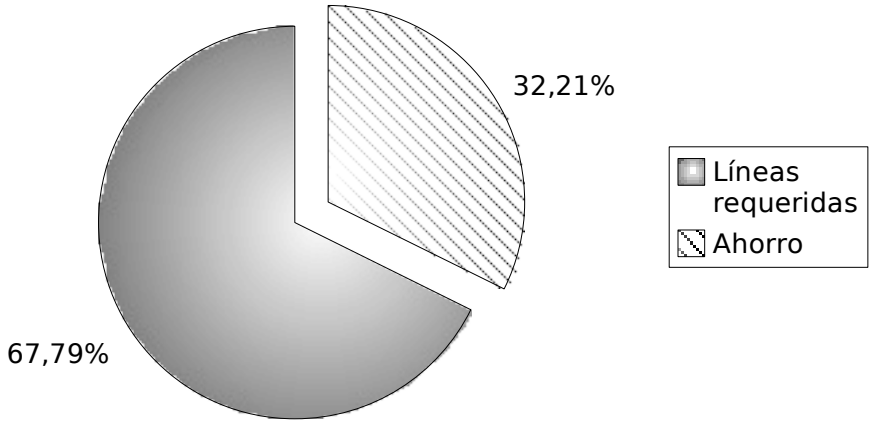


Figura F.9: *Ahorro de código utilizando separación de competencias en la capa transaccional*
Fuente: el autor

F.3.2.4 Demonios

Los demonios del sistema está construida en 30 archivos con un total de 1.991 líneas de códigos conformadas de la siguiente manera:

	Apariciones	*Líneas de código*	*%*
Registro en el logger	*40*	*405*	*20,34 %*
Manejo de error de base de datos	*0*	*0*	*0 %*
Resto de la funcionalidad	*N/A*	*1.586*	*79,66 %*
	Total	1.991	100,00 %

Tabla F.6: *Distribución del código de los demonios*
Fuente: el autor

De haberse realizado la separación de competencias de la 1.991 líneas de código solo serían necesarias 81,67 % de las líneas originales (1.626, la funcionalidad más la aplicación del aspecto de registro en logger – sin contar lo que este ocupa –).

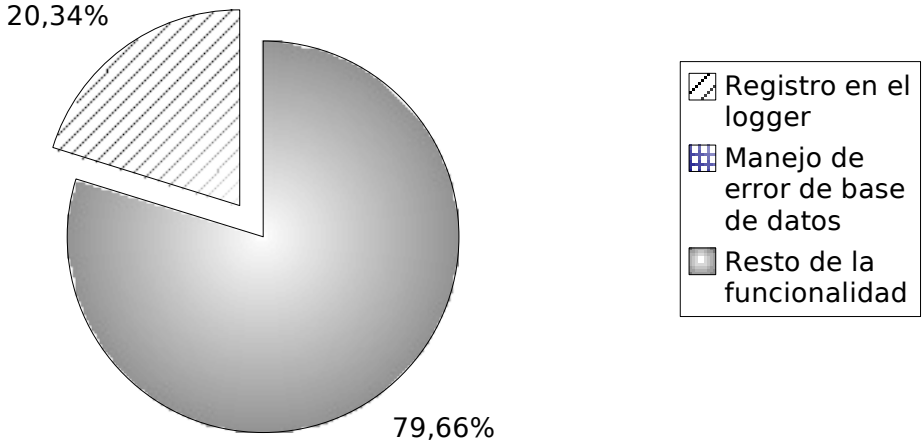


Figura F.10: *Distribución del código de los demonios*
Fuente: el autor

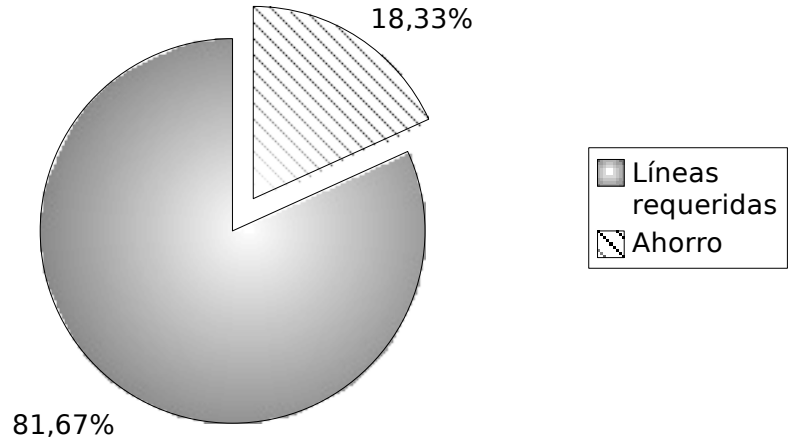


Figura F.11: *Ahorro de código utilizando separación de competencias en los demonios*
Fuente: el autor

F.3.2.5 Otros componentes

Los demonios del sistema está construida en 38 archivos con un total de 5.893 líneas de códigos conformadas de la siguiente manera:

	Apariciones	*Líneas de código*	*%*
Registro en el logger	*7*	*70*	*1,19 %*
Manejo de error de base de datos	*0*	*0*	*0 %*
Resto de la funcionalidad	*N/A*	*5.823*	*79,66 %*
	Total	5.893	100,00 %

Tabla F.7: *Distribución del código de los otros componentes*
Fuente: el autor

De haberse realizado la separación de competencias de la 5.893 líneas de código solo serían necesarias 98,93 % de las líneas originales (5.830, la funcionalidad más la aplicación del aspecto de registro en logger – sin contar lo que este ocupa –).

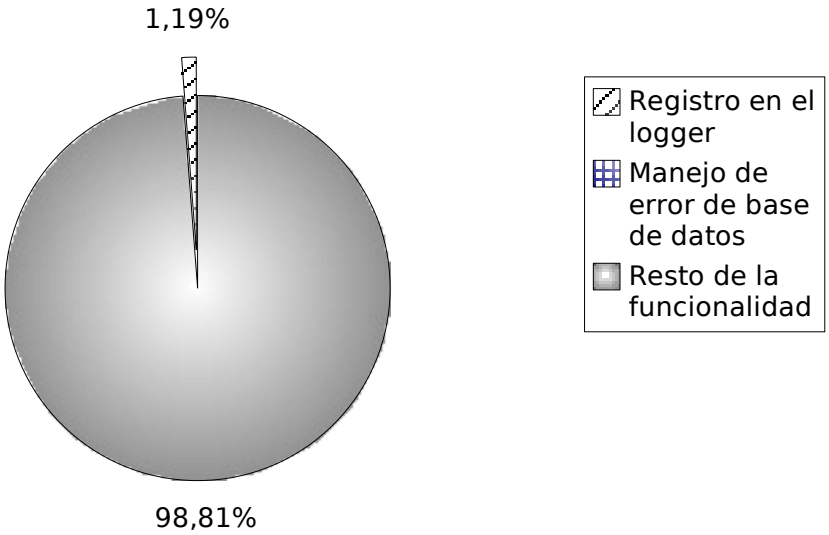


Figura F.12: *Distribución del código de los otros componentes*
Fuente: el autor

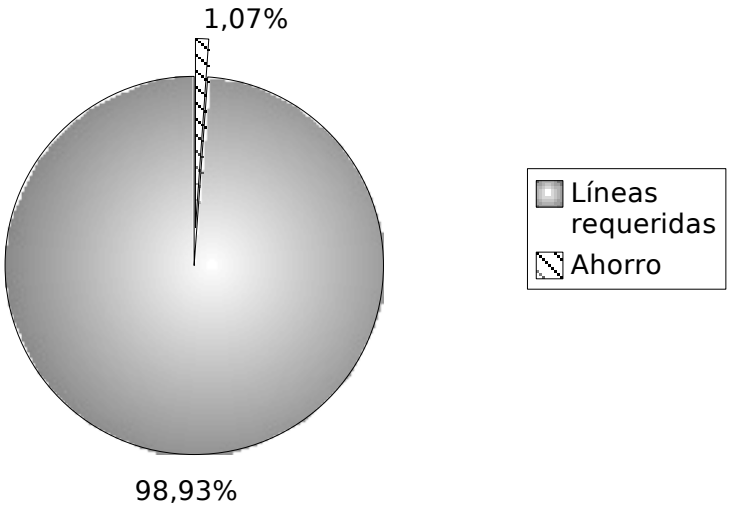


Figura F.13: *Ahorro de código utilizando separación de competencias en los otros componentes*
Fuente: el autor

F.3.2.6 El sistema completo

El sistema SGEM está construido en 400 archivos con un total de 48.786 líneas de códigos conformadas de la siguiente manera:

	Apariciones	*Líneas de código*	*%*
Registro en el logger	*883*	*8.870*	*2,09 %*
Manejo de error de base de datos	*102*	*1.020*	*18,18 %*
Resto de la funcionalidad	*N/A*	*38.896*	*79,73 %*
	Total	48.786	100,00 %

Tabla F.8: *Distribución de código del sistema completo*
Fuente: el autor

De haberse realizado la separación de competencias de la 48.786 líneas de código solo serían necesarias 81,75 % de las líneas originales (39.881, la funcionalidad más la aplicación de cada uno de los aspectos – sin contar lo que ocupan los aspectos –).

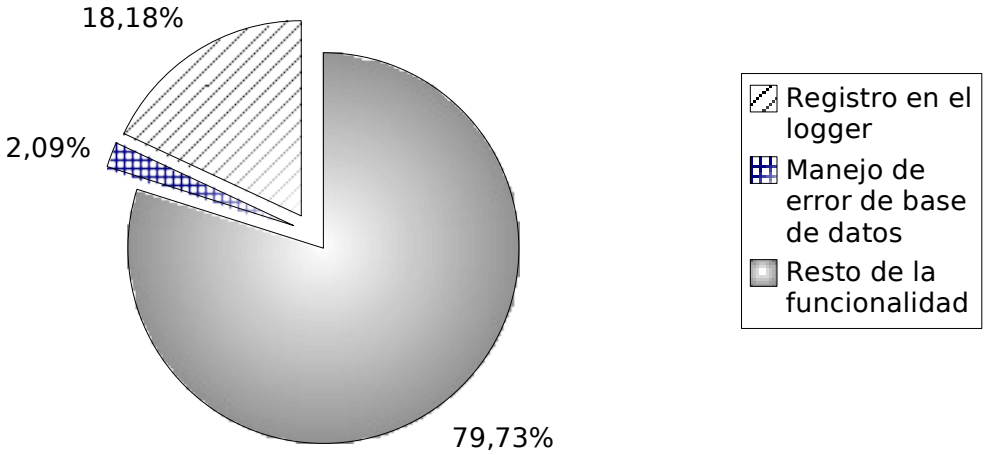


Figura F.14: *Distribución de código del sistema completo*
Fuente: el autor

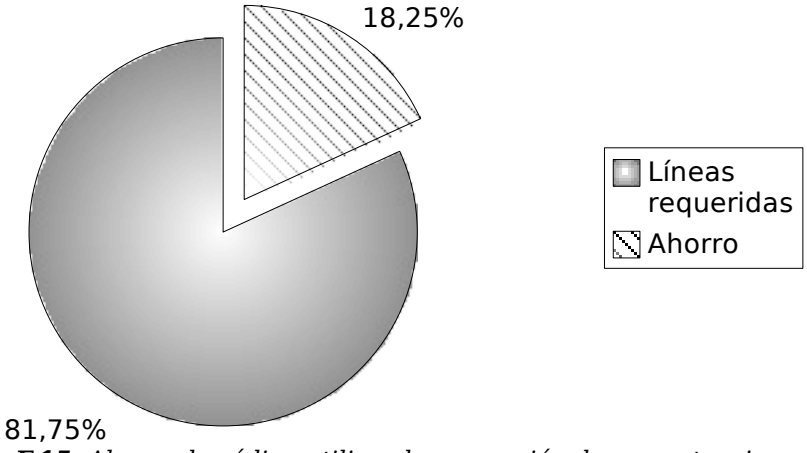


Figura F.15: *Ahorro de código utilizando separación de competencias en el sistema completo*
Fuente: el autor

Apéndice G

Diagramas de clases de gmcs

A continuación se presentas los diagramas de clases de las secciones más importantes del compilador de C# con soporte para tipos genéricos (`gmcs`) del proyecto Mono:

1. El primer diagrama corresponde a la representación interna del compilador de las expresiones del lenguaje

2. El segundo diagrama corresponde a la representación interna del compilador de las construcciones del lenguaje

3. El tercer diagrama corresponde a la representación interna del compilador de los miembros (de clases, estructuras, interfaces y espacios de nombres) del lenguaje

Nota: El los diagramas no aparecen paquetes debido a que todo el código del compilador `gmcs` se encuentra en un único espacios de nombres: `Mono.CSharp`

(Véase el documento "G1 Expresiones.svg" que se encuentra dentro de este documento PDF como archivo adjunto)

(Véase el documento “G2 Construcciones.svg” que se encuentra dentro de este documento PDF como archivo adjunto)

(Véase el documento “G3 Miembros.svg” que se encuentra dentro de este documento PDF como archivo adjunto)

Apéndice H

Diagrama de clase de las nuevas construcciones

A continuación se presentas el diagrama de clases de las clases que sirven de apoyo en el proceso de traducción de las nuevas construcciones en código entendible por el compilador `gmcs`

Nota: El los diagramas no aparecen paquetes debido a que todo el código se encuentra en un único espacios de nombres: `Mono.CSharp` (el espacio de nombres del compilador `gmcs`)

(Véase el documento “H Implementacion.svg” que se encuentra dentro de este documento PDF como archivo adjunto)

Apéndice I

Software utilizado

A continuación se listan el Software utilizado para la elaboración de este trabajo especial de grado:

- La versión de mono modificada es la versión 1.2.4
- Se trabajó utilizando Ubuntu 6.10 y 7.04 como sistema operativo
- El entorno de desarrollo utilizado es Mono Develop
- La herramienta de generación de diagramas utilizada es Visual Paradigm for UML Community Edition sobre Ubuntu 7.04
- La herramienta de ingeniería inversa utilizada para examinar la estructura del compilador para C# de Mono es Visual Paradigm for UML Enterprise Edition con una licencia de evaluación sobre Microsoft Windows XP con Microsoft .Net SDK 2.0
- El sistema de control de versiones utilizado es subversion
- La herramienta para el retoque de los diagramas utilizada es Inkscape (disponible en los repositorios de Ubuntu)
- Para la elaboración del tomo se utilizó Open Office, la tipografía

utilizada es: Gentium y DejaVu Sans Mono que son provistas de serie por Ubuntu

- Para poder instalar las versiones más recientes de Mono y Mono Develop fue necesario agregar los repositorios “http://debian.meebey.net/ ./” y “http://directhex.mfgames.com/ ./” a los orígenes de software de Ubuntu 7.04
- Fue necesario instar manualmente (debido a que se decidió utilizar una versión más reciente a la provista de serie por Ubuntu) los paquetes libgnome-vfs2.0-cil_2.16.0-6_i386.deb y libgtkhtml2.0-cil_2.16.0-6_i386.deb (tomados de los repositorios propuestos para Ubuntu 7.10)
- Los paquetes que se necesitaron instalar son: libglib2.0-0 , pkg-config , autoconf , automake , bison , g++ , libtool , make , python , libglib2.0-dev, subversion, monodevelop, monodevelop-versioncontrol y nautilus-script-collection-svn

Apéndice J

Guías de programación orientada a aspectos

A continuación se presentan una serie de guías que permiten ahondar en la programación orientada a aspectos (POA), en ellas se estudia la POA en general y dos lenguajes de aspectos: Hiper/J y AspectJ. Estas guías son extractos de el proyecto de fin de carrera de Juan Manuel Nieto Moreno* para obtener el título de ingeniero informático en la Universidad de Sevilla (España), presentado en el año 2003, titulado "Programación orientada a aspectos aplicada a tecnologías WEB" y tutelado por Antonia M. Reina Quintero [Nieto, 2003]. Las guías presentadas son:

- Introducción a la Programación Orientada a Aspectos
- Introducción a Hyper/J y la Separación Multidimensional de Competencias
- AspectJ en la Programación Orientada a Aspectos
- Guía rápida de referencia de AspectJ

* Se le agradece a Juan Manuel Nieto Moreno por haber autorizado la inclusión de este material como complemento a este trabajo especial de grado

Introducción a la Programación Orientada a Aspectos

Juan Manuel Nieto Moreno *
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT
La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA es usado en [1] para referirse a varias tecnologías relacionadas como los métodos adaptivos, los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

1. INTRODUCCIÓN

Muchas veces nos encontramos, a la hora de programar, con problemas que no podemos resolver de una manera adecuada con las técnicas habituales usadas en la programación procedural o en la orientada a objetos. Con éstas, nos vemos forzados a tomar decisiones de diseño que repercuten de manera importante en el desarrollo de la aplicación y que nos alejan con frecuencia de otras posibilidades. Por otro lado, la implementación de dichas decisiones a menudo implica escribir líneas de código que están distribuidas por toda, o gran parte, de la aplicación para definir la lógica de cierta propiedad o comportamiento del sistema, con las consecuentes dificultades de mantenimiento y desarrollo que ello implica. En inglés este problema se conoce como *tangled code*, que podríamos traducir como código enredado. El hecho es que hay ciertas decisiones de diseño que son difíciles de capturar con las técnicas antes citadas, debiéndose al hecho de que ciertos problemas no se dejan encapsular de igual forma que los que habitualmente se han resuelto con funciones u objetos. La resolución de éstos supone o bien la utilización de repetidas líneas de código por diferentes componentes del sistema, o bien la superposición dentro de un componente de funcionalidades dispares. La programación orientada a aspectos, permite, de una manera comprensible y clara, definir nuestras aplicaciones considerando estos problemas. Por aspectos se entiende dichos problemas que afectan a la aplicación de manera horizontal y que la programación orientada a aspectos persigue poder tenerlos de manera aislada de forma adecuada y comprensible, dándonos la posibilidad de poder construir el sistema componiéndolos junto con el resto de componentes.

La programación orientada a objetos (POO) supuso un gran paso en la ingeniería del software, ya que presentaba un modelo de objetos que parecía encajar de manera adecuada con los problemas reales. La cuestión era saber descomponer de la mejor manera el dominio del problema al que nos enfrentáramos, encapsulando cada concepto en lo que se dio en llamar objetos y haciéndoles interactuar entre ellos, habiéndoles dotado de una serie de propiedades. Surgieron así numerosas metodologías para ayudar en tal proceso de descomposición y aparecieron herramientas que incluso automatizaban parte del proceso. Esto no ha cambiado y se sigue haciendo en el proceso de desarrollo del software. Sin embargo, frecuentemente la relación entre la complejidad de la solución y el problema resuelto hace pensar en la necesidad de un nuevo cambio. Así pues, nos encontramos con muchos problemas donde la POO no es suficiente para capturar de una manera clara todas las propiedades y comportamientos de los que queremos dotar a nuestra aplicación. Así mismo, la programación procedural tampoco nos soluciona el problema.

---

* Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

Entre los objetivos que se ha propuesto la POA están, principalmente, el de separar conceptos y el de minimizar las dependencias entre ellos. Con el primer objetivo se persigue que cada decisión se tome en un lugar concreto y no diseminada por la aplicación. Con el segundo, se pretende desacoplar los distintos elementos que intervienen en un programa. Su consecución implicaría las siguientes ventajas:

- Un código menos enmarañado, más natural y más reducido.
- Mayor facilidad para razonar sobre los conceptos, ya que están separados y las dependencias entre ellos son mínimas.
- Un código más fácil de depurar y más fácil de mantener.
- Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.
- Se tiene un código más reusable y que se puede acoplar y desacoplar cuando sea necesario.

2. ¿NECESITAMOS ASPECTOS?

En muchas situaciones la programación orientada a objetos deja de ser útil y se convierte en un proceso insostenible. Por un lado, hay modelos orientados a objetos que están muy especializados en un determinado dominio de problemas y que no son convenientes para los demás. También hay cuestiones a resolver que no están vinculadas con ningún problema en particular, sino que nos las encontramos por toda la aplicación, afectando a gran parte de ella y que nos obligan a escribir el mismo código por muchos sitios diferentes para poder solucionarlas. Este es el código que hace que nuestra aplicación orientada a objetos bien diseñada vaya siendo cada vez menos legible y delicada.

Por otro lado, están las interfaces de los objetos, clara debilidad en la evolución de los mismos, ya que deben ser definidas desde el principio. Mientras las necesidades del problema no cambien, el diseño con interfaces funciona bien. Pero eso no es lo habitual, sino que los negocios cambian, lo hace el dominio o bien aparecen nuevas necesidades. En tal caso, hay que cambiar las interfaces y ello supone también cambiar mucho código en todas las clases que las implementan. Para realizar tales cambios, el programador se ve obligado a analizar detalladamente la implementación actual, navegando por las diferentes clases para obtener una visión global del problema al que se enfrenta. Veremos en el ejemplo que acompaña a este documento en qué pueden ayudarnos los aspectos y si ciertamente la orientación a aspectos traerá tantos beneficios como los más optimistas auguran.

3. HISTORIA

Muchos observan la programación orientada a aspectos como el siguiente paso en la evolución de la ingeniería del software e intentan hacer ver como ésta supondrá, como lo hizo en su momento la POO, un cambio importante en el diseño de aplicaciones. No obstante, recordemos que fue sobre 1960 cuando apareció la primera implementación usable de los conceptos de orientación a objetos con el lenguaje Simula-68, mientras que no fue hasta 1980, veinte años después, cuando de verás se empezaron a usar de manera comercial en proyectos reales, dichas técnicas, es decir, con mucho dinero por medio. Fue entonces el *boom* de las telecomunicaciones y el crecimiento de C++ frente a C y COBOL.

El concepto de POA fue introducido por Gregor Kiczales y su grupo, aunque el equipo Demeter [3] había estado utilizando ideas orientadas a aspectos antes incluso de que se acuñara el término. El trabajo del grupo Demeter estaba centrado en la programación adaptiva, que puede verse como una instancia temprana de la POA. Dicha metodología de programación se introdujo alrededor de 1991. Consiste en dividir los programas en varios bloques de cortes. Inicialmente, se separaban la representación de los objetos del sistema de cortes; luego se añadieron comportamientos de estructuras y estructuras de clases como bloques constructores de cortes. Cristina Lopes propuso la sincronización y la invocación remota como nuevos bloques [4]. No fue hasta 1995 cuando se publicó la primera definición temprana del concepto de aspecto, realizada también por el grupo Demeter y que sería la siguiente:

"Un aspecto es una unidad que se define en términos de información parcial de otras unidades".

Por suerte las definiciones cambian con el tiempo y se hacen más comprensibles y precisas. Actualmente es más apropiado hablar de la siguiente definición de Gregor Kiczales:

"Un aspecto es una unidad modular que se dispersa por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa".

4. UN PROCESADOR DE IMÁGENES

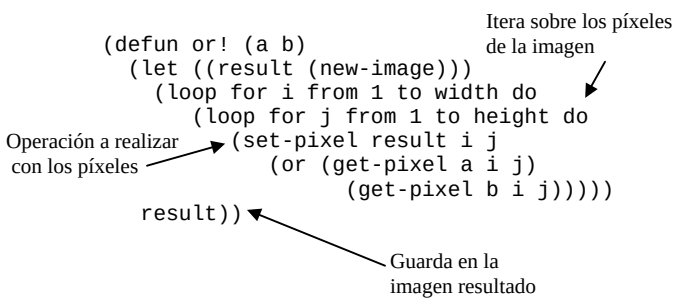
El siguiente ejemplo está sacado del trabajo publicado en 1997 para la *European Conference on Object-Oriented Programming* (ECOOP) celebrada en Finlandia [2]. Supongamos una aplicación de procesamiento de imágenes en blanco y negro. El dominio del problema son las imágenes que deben pasar por una serie de filtros para obtener el resultado deseado. Se podrían hacer tres implementaciones: una que fuese fácil de entender pero ineficiente, una eficiente pero difícil de entender y una tercera, basada en POA, que a la vez sería fácil de entender y eficiente.

Figura 1 Procesador de imágenes

Los objetivos entonces a cumplir para el desarrollo del procesador de imágenes van a ser que sea fácil de desarrollar y de mantener, así como que haga un uso eficiente de la memoria. Los motivos son evidentes: con el primero conseguiremos de forma rápida y libre de fallos futuras mejoras del sistema; el segundo se debe al tamaño de las imágenes, que suele ser elevado, por lo que conviene minimizar el uso de memoria y los requerimientos de almacenamiento.

4.1 DEFINICIÓN DEL PROBLEMA

Aún con la tradicional programación procedural podemos implementar de una forma clara, concisa y apropiada para el dominio del problema nuestro procesador de imágenes, consiguiendo el primer objetivo. De esta forma los filtros pueden ser definidos como procedimientos que toman una o varias imágenes de entrada y producen una imagen de salida. Se tendrían así un conjunto de procedimientos para los filtros más básicos y que servirían para definir los filtros más complicados en base a éstos. Así por ejemplo, un filtro `or!` que tomara dos imágenes y que devolviera una resultado de aplicar esta operación sería implementado en Common Lisp como sigue:



Empezando con este sencillo filtro `or!` y junto con otras primitivas, el programador puede construir filtros más complejos en base a ellos. Así por ejemplo podemos hacer:

– Un filtro que se quede con la parte superior de una imagen:[1]

```
(defun down! (a)
  (let ((result (new-image)))
    (loop for i from 1 to width do
      (loop for j from 1 to height-1 do
   (set-pixel result i j
           (get-pixel a i j+1))))
   result))
```

– Un filtro que tome dos imágenes y devuelva otra con la diferencia entre ellas:

```
(defun remove! (a b)
    (and! a (not! b)))
```

– Un filtro que se quede con la parte superior de una imagen:

```
(defun top-edge! (a)
     (remove! a (down! a)))
```

– Un filtro que se quede con la parte inferior de una imagen:

```
(defun botton-edge! (a)
     (remove! a (up! a)))
```

– Un filtro que se quede con los píxeles negros de las partes superior e inferior:

```
(defun horizontal-edge! (a)
     (or! (top-edge! a)
          (botton-edge! a)))
```

Como puede verse únicamente los filtros más básicos hacen uso de las estructuras `loop` para iterar sobre los píxeles de las imágenes. Los filtros de más alto nivel, tal como `horizontal-edge!`, se expresan en términos de los filtros básicos. El código resultante es fácil de leer, comprender, depurar y extender. Se cumple entonces el primero de los objetivos.

4.2 OPTIMIZANDO LA MEMORIA

La implementación anterior no tiene en cuenta nuestro segundo objetivo de optimización de los recursos de memoria. Cada vez que se llama a un procedimiento, se itera sobre una serie de imágenes y se produce una nueva imagen resultante. Se crean demasiadas nuevas imágenes que en muchas ocasiones sólo sirven de resultados intermedios, lo que conlleva excesivas referencias a memoria y petición de espacio para almacenar, que implica además fallos de caché, fallos de página y todo ello un bajo rendimiento.

[1] La sintaxis de este ejemplo no es Lisp estándar con objeto de que sea más legible.

Para solucionarlo se puede tomar una perspectiva más global del problema, tomando aquellos resultados intermedios como entradas de otros filtros y programar una versión que sintetice los bucles de cada uno de los filtros de manera apropiada para implementar la funcionalidad original, creando el menor número posible de imágenes intermedias. Así tendríamos el siguiente código para el mismo procedimiento anterior `horizontal-edge!`:

```
(defun horizontal-edge! (a)
 (let ((result (new-image))
   (a-up (up! a))
   (a-down (down! a)))
  (loop for i from 1 to width do
   (loop for j from 1 to height do
     (set-pixel result i j
       (or (and (get-pixel a i j)
                (not (get-pixel a-up i j))
           )
           (and (get-pixel a i j)
                (not (get-pixel a-down i j))
           )
  ))))
 result))
```

Comparado con el original, la lógica de este método es mucho más confusa y como se dice en la terminología inglesa, el código está enredado (*tangled*). Lo que se ha hecho es incorporar en un solo método todos los anteriores, eliminando así algunas de las iteraciones, fusionando las operaciones `and!` y `or!`.[2] Y aunque se ha mejorado la eficiencia, esto ha conllevado la pérdida de la clara estructura original.

En un pequeño ejemplo como este, aún podríamos sobrellevar el deterioro de la claridad que ha supuesto esta modificación. Pero en aplicaciones reales la complejidad debida a este tipo de optimizaciones termina por ser un verdadero inconveniente a la hora de seguir desarrollando o manteniendo el código. Este ejemplo proviene de una aplicación real de reconocimiento de caracteres, donde este método es parte de un subcomponente importante. Implementándolo de una manera sencilla, pero ineficiente, se necesitan 768 líneas de código, mientras que la versión optimizada, con la fusión de los bucles, manteniendo resultados intermedios, asignación de memoria en tiempo de compilación y estructuras de datos intermedias para mejorar las prestaciones requiere 35213 líneas de código, que además están totalmente enredadas. Esta implementación es extremadamente difícil de mantener, ya que pequeños cambios en la funcionalidad requieren entender todo el enredamiento de código que se ha empleado. Pero es la única que es práctica debido a las prestaciones. Las figuras 2 y 3 muestran dos diferentes diagramas de la versión sin optimizar del filtro `hotizontal-edge!` La figura 2 presenta una descomposición funcional que se corresponde directamente con el dominio del problema. La 3 es un diagrama de flujo, donde las cajas representan los filtros básicos y las aristas el flujo de datos entre ellos en tiempo de ejecución. Abajo, la caja etiquetada con `a`, es la imagen de entrada.

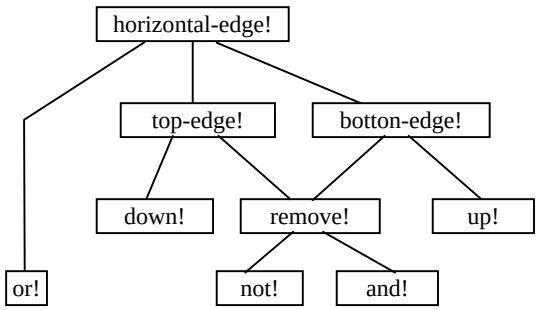


Figura 2 Descomposición funcional de horizontal-edge!

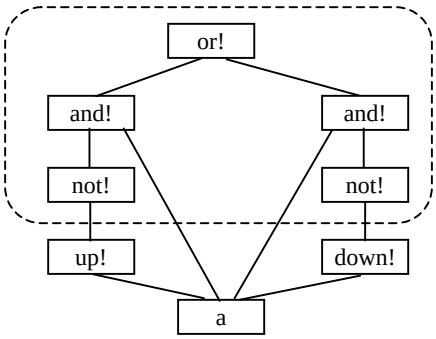


Figura 3 Diagrama de flujo de horizontal-edge!

4.3 CÓMO SE ENTRELAZA EL CÓDIGO

La figura anterior nos da otro punto de vista para entender porqué se enreda el código en la aplicación del ejemplo. A la izquierda se ha presentado la estructura jerárquica de la funcionalidad del filtro. A la derecha, un diagrama de flujo de los datos en la versión original sin optimizar de `horizontal-edge!` En el diagrama, las cajas y líneas muestran los filtros básicos y el flujo de datos entre ellos. La caja ovalada de línea discontinúa muestra la parte que se fusionó en un único bucle en la versión optimizada.

Como se ve, la caja ovalada no incorpora todas las acciones de `horizontal-edge!`. De hecho, no se corresponde con ninguna de las unidades funcionales jerárquicas de la izquierda. Mientras que las dos propiedades que se quieren implementar, la funcionalidad y la fusión de los bucles, ambas provienen de los mismos filtros básicos, tienen que combinarse de diferente forma cuando se fusionan los filtros. La funcionalidad sigue un orden jerárquico a la manera tradicional. La fusión de los bucles se realiza con aquellos filtros cuya estructura cíclica tiene la misma forma y que además, son vecinos

[2] Si se fusionaran también los loops de up! y down! sería entonces cuando pasaría a ser totalmente ilegible. Veremos como la versión con POA sí los fusionará.

directos en el diagrama de flujo de datos. Cada una de estas reglas de composición es fácil de entender viendo cada una de los dibujos anteriores por separado. Pero la relación de composición entre ambas es bastante más complicada y es difícil de observar la manera de componerse de una en el dibujo del otro.

La confusa relación anterior es la que provoca que el código se entremezcle y se debe al sencillo mecanismo de composición que nos ofrecen los lenguajes actuales: llamadas a procedimientos. Este es de utilidad para implementar versiones ineficientes, pero una vez hecho ésto, debido a las diferentes reglas de composición que requieren una intención y otra, nos obligan a combinar el código de una manera artificiosa y como se diría, a mano, terminando en lo que ya hemos dado en llamar código entrelazado o *tangled* en inglés.

En general, cuando se quieren coordinar dos propiedades que afectan de diferente forma al código, es cuando se dice que ambas se entrelazan. Debido a que los lenguajes basados en procedimientos, ya sean orientados a objetos, funcionales o procedurales soportan un solo mecanismo de composición, el programador tiene que hacer la composición manualmente, conllevando esto a la complejidad del código y al código enredado. Esto nos lleva a la definición de dos términos que ayudarán a entender la definición de aspecto. Respecto a una aplicación y a su implementación usando un lenguaje basado en procedimientos, una propiedad se implementará como:

- *un componente*, si puede ser encapsulado de forma clara en un procedimiento, como pueden serlo los objetos, métodos, procedimientos, API... Se entiende de forma clara, está bien localizada, es de fácil acceso y que se deja componer como sea necesario. Los componentes suelen ser unidades de la descomposición funcional del sistema, como filtros de imágenes, la cuenta de un banco o los componentes de las interfaces gráficas.

- *un aspecto*, si no se puede encapsular de forma clara en un procedimiento. Los aspectos no suelen ser unidades funcionales que obtengamos al descomponer el sistema, sino más bien, propiedades que afectan al rendimiento o el comportamiento de los componentes. Como ejemplos de aspectos, podemos citar: patrones de acceso a memoria, sincronismo de objetos concurrentes, control de errores y manejo de excepciones, utilización de caché...

Con estos términos claros podemos ahora establecer cual es el objetivo de la programación orientada a aspectos: proporcionar al programador una técnica para, de una forma clara, separar componentes y aspectos unos de otros[3], dotando de mecanismos que hacen posible abstraer estos para después componerlos dando resultado al sistema final. Mientras que expresado con las mismas palabras, diremos que el objetivo de los lenguajes basados en procedimientos es proporcionar al programador una técnica para separar componentes de componentes (no hay posibilidad de definir aspectos), dotando de mecanismos que hacen posible abstraer estos para después componerlos dando resultado al sistema final.

4.4 CÓMO HACERLO CON POA

Ya que hemos visto la necesidad y utilidad de separar los componentes de los aspectos, podemos pasar a definir una nueva implementación del procesador de imágenes basado en POA. El objetivo de esta sección será explicar cómo es la estructura de una implementación basada en POA, pero no explicarla completamente.

La estructura de una aplicación cuya implementación se basa en POA es análoga a la estructura de una que se base en un lenguaje procedural. Podemos decir que esta segunda consiste en: (i) un lenguaje, (ii) un compilador para dicho lenguaje y (iii) un programa escrito en el anterior lenguaje. Por otro lado, una aplicación orientada a aspectos consiste en: (i.a) un lenguaje de componentes, con el que programar los componentes; (i.b) uno o más lenguajes de aspectos con los que programar los aspectos; (ii) una herramienta para combinar los aspectos y los componentes (*aspect weaver*); (iii.a) un programa que implemente los componentes usando el lenguaje de componentes; (iii.b) uno o más programas que implementen los aspectos usando los lenguajes de aspectos. El proceso de entrelazado de código dependiendo del lenguaje y la herramienta de entrelazado, puede hacerse en tiempo de ejecución (*run-time weaving*) o en tiempo de compilación (*compile-time weaving*).

4.5 EL LENGUAJE DE COMPONENTES

Para este ejemplo utilizaremos un lenguaje de componentes y uno de aspectos. El de componentes es similar al usado para desarrollar la versión sin aspectos anterior, sólo que se le han introducido unos cambios que permiten ver de forma clara cómo trabajan los aspectos. Así los cambios son los siguientes:

- Los filtros no son procedimientos explícitamente.
- Los bucles se definen de otra forma, de manera que la estructura del bucle sea mucho más explícita.

Con tales cambios, el mismo filtro anterior `or!` se escribe de la siguiente forma:

[3] Componentes de componentes, aspectos de aspectos y componentes de aspectos.

```
(define-filter or! (a b)
  (pixelwise (a b)
             (pixel-from-a pixel-from-b)
             (or pixel-from-a pixel-from-b)
  )
)
```

El constructor `pixelwise` es un iterador, que en este caso itera sobre cada uno de los elementos de las imágenes `a` y `b`, asignando en `pixel-from-a` y `pixel-from-b` cada uno de los elementos que recorre y aplicándoles después la operación que sigue, sea en este caso la or, devolviendo finalmente una imagen con el resultado. De igual forma se construyen los otros filtros que implementan las funciones de agregación, diferencia y desplazamiento de píxeles necesarias para la aplicación. El hecho de introducir este constructor para los bucles posibilita que a continuación, el lenguaje de aspectos sea capaz de detectar, analizar y fusionar los bucles de forma mucho más fácil.

4.5 EL LENGUAJE DE ASPECTOS

El diseño del lenguaje de aspectos para este ejemplo se basa en la observación y las conclusiones que se tienen del gráfico del diagrama de flujo de datos, ya que ahí se entiende cómo se pueden fusionar los bucles. El lenguaje de aspectos es un sencillo lenguaje procedural que permite simples operaciones sobre los nodos del diagrama de flujo de datos. Así el programa de aspectos puede detectar los bucles que pueden fusionarse y proceder a fusionarlos. El siguiente fragmento de código es una parte de la aplicación donde se lleva a cabo la fusión de los bucles que tiene la misma estructura y que están seguidos en el diagrama de flujo de datos. Primero comprueba las parejas de nodos del grafo conectadas por una arista que tengan la misma estructura `pixelwise`, esto es la que tiene un `or` o un `and`. Aquellas que encuentre las fusiona en un único bucle, con la misma estructura y que combina de manera adecuada las entradas, las variables del bucle y el cuerpo de los dos bucles originales. Véase el siguiente fragmento:

```
(cond ((and (eq (loop-shape node) ‘pointwise)
            (eq (loop-shape
input)‘pointwise))
       (fuse loop input ‘pointwise
             :inputs (splice …)
             :loop-vars (splice …)
       :body (subst …))))
```

Describir las reglas de composición y fusión para los cinco tipos de bucles que se dan en la aplicación real requiere del orden de una docena de cláusulas similares para indicar cuando y cómo realizarla. Es por ello que no es posible pretender que un compilador haga por sí solo esta clase de optimizaciones, así como por ejemplo compartir resultados intermedios o mantener el uso de la memoria en tiempo de ejecución dentro de unos límites. De momento, los compiladores no han llegado a ese estado.

4.6 PROCESO DE ENTRELAZADO

La aplicación encargada del proceso de combinación de los componentes y los aspectos, en la aplicación real desarrollada en *Xerox Palo Alto Research Center* y que sirve aquí de ejemplo produce como salida un programa en C, tomando como entrada los componentes y los aspectos. El proceso requiere de tres fases como se ilustra en la figura 4.

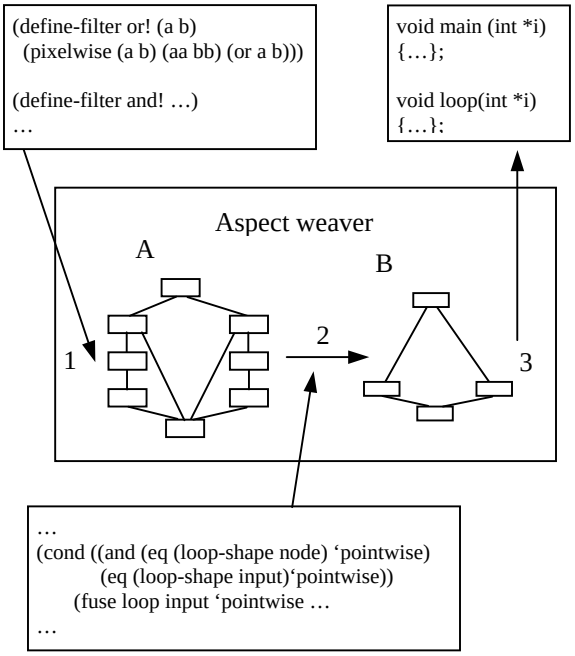


Figura 4 Proceso en tres fases del entrelazador de aspectos (aspect weaver)

En la primera fase el entrelazador de aspectos genera un grafo de flujo de datos con el programa de componentes. En este grafo, los nodos representan filtros básicos y las aristas el flujo de imágenes entre un filtro y otro. Cada nodo contiene un único constructor de bucles como los presentados antes. Así por ejemplo, el nodo etiquetado con A en la figura contiene el siguiente constructor, donde #<...> hace referencia a las aristas que llegan al nodo:

```
(pointwise (#<edge1> #<edge2>) (i1 i2)
   (or i1 i2))
```

En la segunda fase, se utiliza el programa de aspectos para editar el grafo fusionando nodos y ajustando el cuerpo de los filtros de manera adecuada. El resultado es un grafo con menos nodos y donde cada uno de los nuevos nodos tiene ahora más operaciones básicas entre píxeles que antes de esta fase. Por ejemplo, el nodo etiquetado con B, que corresponde a la fusión de cinco bucles del original grafo, tiene el siguiente cuerpo:

```
(pointwise (#<edge1> #<edge2> #<edge3>)
           (i1 i2 i3)
   (or (and (not i1) i2) (and (not i3) i2))))
```

Finalmente, en la tercera de las fases, un sencillo generador de código recorre el nuevo grafo de los nodos fusionados y genera una función en *C* para cada nodo, así como una función `main` que llama a estas funciones en el orden apropiado, pasándoles los resultados a cada una de la función del nodo que le preceda. Esta generación de código es sencilla puesto que cada nodo contiene únicamente un constructor de bucles donde el cuerpo está formado únicamente por funciones básicas entre píxeles.

Un aspecto fundamental de este sistema es que el entrelazador no es necesariamente un compilador muy complejo e "inteligente". Al usar POA hemos dejado que sea el programador quien defina todas las decisiones sobre las estrategias implementación haciendo uso de un lenguaje apropiado de aspectos. El entrelazador no necesita tomar ninguna decisión inteligente. Pedir al programador que se encargue de los aspectos de implementación podría parecer un paso atrás en el arte de programar. Sin embargo, no es tanto así. Cuando por ejemplo un programador está definiendo un aspecto que afecta al uso de la memoria, usar POA supone definir las estrategias de implementación a un nivel apropiado de abstracción, mediante un lenguaje de aspectos adecuado. No se requiere entrar en los detalles de implementación, ni se requiere trabajar directamente con el código confuso enredado. A la hora de evaluar los beneficios de una versión de una aplicación orientada a aspectos es importante compararla con ambas la versión clara e ineficiente y la eficiente pero compleja.

4.7 RESULTADO FINAL

La aplicación real del ejemplo presentado es algo más complicada. Además del aspecto aquí presentado hace uso de dos más: uno para compartir la ejecución de operaciones básicas comunes, reduciendo el volumen de éstas; y otro para asegurar que a la misma vez se tienen en memoria el menor número de imágenes posibles, para optimizar el uso de memoria. Los tres están escritos con el mismo lenguaje de aspectos.

Como hemos visto en el ejemplo, la implementación mediante POA ha conseguido los objetivos propuestos al principio. Se puede razonar fácilmente sobre el código de la aplicación, por lo que es además fácil de mantener y ha sido fácil de desarrollar, mientras que al mismo tiempo es bastante eficiente. Al mantenerse limpio el código de los componentes, es fácil para el programador entenderlos y ver la forma en la que se combinan unos con otros. Igualmente la definición de los aspectos es clara y es sencillo entender como se combinan y su efecto sobre los componentes. Si se introdujeran cambios en los componentes o en el aspecto de fusión estos se reflejarían de manera sencilla sin más que volver a utilizar el entrelazador de código y dejarle hacer su trabajo. Lo laborioso que es escribir los detalles de la implementación se ha eliminado del trabajo del programador, por lo que no tiene que enfrentarse a esos pequeños detalles y minuciosidades que tan complicadas hacen las versiones eficientes.

La versión real de esta aplicación desarrollada en *Xerox Palo Alto Research Center* tiene 1039 líneas de código, contando los componentes y los tres aspectos. El entrelazador de aspectos, incluyendo un componente de generación de código reusable, tiene 3520 líneas[4]. La versión programada a mano requiere 35213 líneas, lo que es una diferencia significativa en tamaño, pensando además que en esta el código es mucho más complejo y delicado, debido a que está todo mezclado: la funcionalidad y las optimizaciones. Eso sí, la versión a mano es más rápida aunque menos eficiente en espacio de memoria. En cualquier caso, la versión orientada a aspectos es unas 100 veces más rápida que la versión sin optimizar.

5. CONCLUSIONES

La separación de conceptos es una herramienta de ingeniería del software que reduce la complejidad de las aplicaciones a niveles manejables para las personas y permite a los desarrolladores centrarse en problemas concretos, ignorando otros que en determinado momento no sean tan importantes. Para hacer un uso efectivo de la separación de conceptos es preciso en cada momento ser capaz de identificar, encapsular, modularizar y manipular diferentes dimensiones o niveles conceptuales de abstracción en cada una de las fases de vida del software, para no verse afectado por los efectos negativos que tiene la dispersión del código, el efecto dominó que pueden suponer cualquier modificación o la necesidad de reestructuración debido a la aparición de nuevos requisitos. Los conceptos de corte suelen aparecer en la fase de captura de requisitos y suelen tomar diferentes formas durante el proceso de desarrollo.

Hasta antes de la POA, los paradigmas de programación se han basado en una única y dominante manera de descomposición del software (clases, funciones, reglas...) La POA persigue la eliminación de lo que se conoce como tiranía de descomposición del software. Actualmente existen diferentes implementaciones que, con más o menos éxito, persiguen este objetivo. La que actualmente goza de mayor difusión es AspectJ [5], una

[4] Realmente la parte del núcleo tiene 1959 líneas. El resto es del componente reusable de generación de código.

extensión orientada a aspectos de Java, en la que se distinguen entre clases, que encapsulan los requisitos funcionales del sistema, y aspectos, que encapsulan los requisitos de corte no funcionales. Otra solución es la aportada por IBM, Hyper/J [6], que soporta la separación e integración de diferentes dimensiones, entendiéndose éstas como diferentes niveles conceptuales de abstracción. Una de ellas sería la dimensión de las clases, donde los requisitos funcionales tendrían cabida. Otra podría ser la de los requisitos no funcionales, que en AspectJ identificaríamos como aspectos (sincronismo, concurrencia,...) y otra podría incluir las restricciones impuestas a la aplicación que desarrollamos (tiempos de ejecución, entornos de funcionamiento,...) AspectJ ha apostado por encapsular las dos dimensiones hasta ahora mejor identificadas. Hyper/J nos deja la puerta abierta a nuevas dimensiones.

6. REFERENCIAS

1. T. Elrad, R. E. Filman, and A. Bader. *Aspect-Oriented Programming*. Commun. ACM, 2001.
2. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John. Irwin, *Aspect-Oriented Programming*, Xerox Palo Alto Research Center, 1997.
3. Karl J. Lieberherr. Adaptive Object-Oriented Software. The Demeter Method. Northeastern University Boston.
4. Cristina Lopes, *A Language Framework for Distributed Programming*, Ph.D.thesis, Northeastern University, noviembre 1997.
5. *The AspectJ™ Programming Guide*, the AspectJ Team, Xerox Parc Corporation
6. Peri Tarr, Harold Ossher. *Hyper/J™ User and Installation Manual*, IBM Research, 2000.

Introducción a Hyper/J y la Separación Multidimensional de Competencias

Juan Manuel Nieto Moreno *
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT
Existen diferentes metodologías de programación que pretenden conseguir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos que las conforman. Gracias a ellas se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Para referirse a estas tecnologías se habla hoy en día en general de programación orientada a aspectos [1], pero cada una de ellas encamina la solución del problema desde puntos de vista diferentes y por ello reciben otros calificativos como métodos adaptivos, filtros de composición, programación orientada a sujetos o separación multidimensional de competencias, a las que nos referimos en este documento. En el contexto de esta última es donde se sitúa la herramienta de IBM Hyper/J, objeto del estudio del presente documento.

1. INTRODUCCIÓN

IBM Hyper/J [2] es una herramienta que soporta la separación e integración multidimensional de competencias en el lenguaje Java. Con esto facilita la adaptación e integración de componentes Java y mejora la modularización y la remodularización no invasiva de las aplicaciones.

La separación de competencias [3] es un intento de descomposición del software en módulos, donde cada uno de los cuales representa y encapsula un problema particular de interés, que denominamos *competencia*. Los lenguajes OO se basan en la descomposición por clases. Pero además de éstas, existen otro tipo de competencias que no pueden ser encapsuladas de tal forma y cuya implementación, con las soluciones actuales, termina distribuida por toda la jerarquía de clases. Eso se debe a que el único tipo de competencia posible son las clases y por ello, otras características como pueden ser la persistencia, la distribución o el sincronismo –a las que llamaremos aspectos– no tienen una representación adecuada. Nos referimos a cada una de estos tipos de competencias –hasta ahora hemos indicado las clases y los aspectos– como dimensiones. La separación de competencias persigue la descomposición del software de acuerdo con una o más de estas dimensiones.

La mayoría de los formalismos actuales soportan la separación de competencias, pero con la tendencia hacia una única dimensión. La descomposición en clases de la POO facilita la evolución de las estructuras de datos, pues están definidas en una única o pocas clases. Pero, por el contrario, impide, o dificulta, la evolución de sus aspectos, pues éstos implican normalmente a varias clases. Usualmente es necesario tener de manera simultánea diferentes dimensiones de abstracción de conceptos, que además es posible que se solapen e interactúen. Se necesita la modularización respecto a diferentes dimensiones, ya sea, por clases, aspectos, roles o cualquier otro criterio. Este problema se conoce como la tiranía de la descomposición dominante (*tyranny of dominant descomposition* [3]), ya que sólo se dispone de una manera de descomponer el software y que además no deja lugar a otro tipo de descomposición. Ejemplos de esto son las, ya citadas, clases (en los lenguajes OO), las funciones (en lenguajes funcionales) o las reglas (en los sistemas basados en reglas).

2. SEPARACIÓN MULTIDIMENSIONAL

La solución de IBM para la separación multidimensional de competencias (SMC) se llama *hyperspaces* e Hyper/J es la herramienta que los hace posible en Java. En los siguientes apartados se describen éste y otros conceptos relevantes de la terminología de Hyper/J.

2.1 HYPERSPACES

Los *hyperspaces* o hiperespacios permiten la identificación explícita de cualquier dimensión y

* Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

competencia en cualquier fase del ciclo de vida del software; la encapsulación de dichas competencias; la identificación y manejo de las relaciones entre dichas competencias; y la integración de las mismas. Formalmente un hiperespacio se define como una tupla *(U,M,H)* donde *U* es un conjunto de unidades recogidas en el hiperespacio; *M* es la matriz de competencias que simultáneamente organiza las unidades de *U* de acuerdo con las principales competencias; y *H* es un conjunto de hipermódulos que especifica cómo construir componentes con las unidades de *U*. Una unidad puede ser, por ejemplo, un aserto, una declaración, un diagrama de estados, una clase, una interfaz, requisitos de especificación o cualquier otra entidad que pueda ser descrita en cualquier lenguaje. Se distingue entre unidades primitivas (métodos, variables,...) y unidades compuestas (clases, paquetes, diagramas de colaboración...) Actualmente las unidades que soporta Hyper/J son paquetes, interfaces, clases, operaciones y atributos Java.

2.2 LA MATRIZ DE COMPETENCIAS

Las competencias (del inglés *concerns*) se pueden modelar como predicados, *c*, sobre un conjunto de unidades, *U*. Dichos predicados se usan para especificar cuándo una unidad está implicada en una competencia determinada. El conjunto de unidades implicadas en una competencia *c* puede representarse por: $U(c)=\{u \in U/c(u)\}$. Si sus conjuntos de unidades no son disjuntos las competencias pueden solaparse. Para organizar las unidades del hiperespacio, éstas pueden entenderse dispuestas en una matriz multidimensional (*concern matrix*), donde cada eje representa una dimensión de competencias y cada punto en un eje una competencia en esa dimensión. De esta manera quedan explícitamente representadas las dimensiones de interés, las competencias que pertenecen a cada dimensión y las unidades implicadas en cada competencia.

Formalmente dicha matriz de competencias se puede definir sobre un conjunto de unidades *U* como una tupla *(C, D)* donde *C* es un conjunto de competencias y *D* un conjunto de dimensiones de éstas, de modo que se cumple que:

- cada competencia en *C* está en una sola de las dimensiones de *D*; y
- las dimensiones de *D* particionan *U*.

De esta forma, cada unidad está involucrada solamente en una competencia en cada dimensión y sus coordenadas pueden entenderse como sus competencias asociadas. Si todas las unidades de la matriz están asociadas con alguna competencia de una misma dimensión, dicha dimensión particionará de manera natural *U*. En caso de no ser así, por conveniencia, en cada dimensión *d* está definida la competencia nula (*None concern*), N_d, que contiene las unidades que no son de interés para dicha dimensión.

2.3 HYPERSLICES

La matriz de competencias identifica las competencias y organiza las unidades de acuerdo con ellas y las dimensiones. Sin embargo, la matriz no soporta, en sí misma, la encapsulación de las competencias (sin ningún otro mecanismo, el conjunto de unidades no puede manejarse como módulos). Así surgen los *hyperslices* o hipersecciones, conjunto de competencias que son declarativamente completos, lo que significa que deben declarar todo aquello que referencian. La completitud de las declaraciones es importante ya que elimina el acoplamiento entre las hipersecciones. No se necesita dar la definición completa de las declaraciones, sino que se pueden utilizar declaraciones abstractas. En lugar de utilizar referencias a otras hipersecciones, el uso de las declaraciones abstractas favorece la reusabilidad e independencia de las hipersecciones y es vital para limitar el impacto de los cambios. Así un conjunto de unidades puede ser tratado como una hipersección, siempre y cuando sea declarativamente completo, y las competencias pueden ser encapsuladas en hipersecciones. Por tanto, independientemente de las limitaciones del lenguaje y la competencia, es posible sintetizar en una hipersección aquellas unidades necesarias para la competencia (más algunas declaraciones abstractas).

Formalmente las hipersecciones se definen como una competencia declarativamente completa $hs \in C$ (*C* es el conjunto de competencias). Así, si una unidad $u_1 \in hs$ referencia de algún modo a otra $u_2 \in U$ entonces $u_2 \in hs$. Suponer, por ejemplo, que una hipersección *X* contiene una unidad u_1 para su método *x()*, el cual usa una función *y()* de su unidad u_2 definida en la hipersección *Y*. Para hacer a *X* declarativamente completa, deberá contener otra unidad u_{2decl}, la cual declare *y()*, sin necesidad de implementarla, de tal forma que u_1 use u_{2decl} en lugar de u_2. Esto elimina el acoplamiento entre *X* e *Y*, entendiendo que u_{2decl} estará finalmente ligada a una implementación en cualquier otra hipersección.

2.4 HYPERMODULES

Los *hypermodules* o hipermódulos incluyen un conjunto de hipersecciones que se pueden componer entre sí, junto con un conjunto de reglas de composición que especifican como han de hacerlo. Los hipermódulos se pueden anidar, lo que permite modelar con ellos componentes funcionales o incluso aplicaciones completas. Las reglas indican las unidades de las diferentes hipersecciones que describen los mismos conceptos y cómo éstas han de integrarse. La sintaxis general para los hipermódulos en Hyper/J es la siguiente:

```
hypermodule hypermoduleName
hyperslices:
  dimensionName1.concernName1,
  dimensionName2.concernName2,
  ...

relationships:
  mergeByName
| nonCorrespondingMerge
| overrideByName;

reglas específicas
end hypermodule;
```

Las dos secciones fundamentales son las correspondientes a la definición de las hipersecciones que intervienen y la definición de las reglas (generales y específicas) de composición de las mismas.

Formalmente un hipermódulo se describe como una tupla *(HS, CR)* donde *HS* es un conjunto de hipersecciones y *CR* es un conjunto de relaciones de composición. Cada una de las reglas se describe como una tupla *(I, r, f, o)* donde *I* es una tupla de unidades de entrada de las hipersecciones del conjunto *HS*; *r* es una relación de correspondencia que define cómo las unidades en *I* se interrelacionan; *f* es una función de composición $f : (I \times r) \rightarrow U$, que indica cómo componer las unidades en *I* de acuerdo con *r*; y *o* es la unidad e salida que se obtiene usando *f*.

2.5 RELACIONES DE COMPOSICIÓN

Las hipersecciones pueden solaparse y compartir unidades o estar interrelacionadas por una o más relaciones de integración que indican cómo han de combinarse. Podemos identificar dos clases distintas de relaciones: las dependientes del contexto y las independientes del contexto. Las relaciones con solapamiento son un ejemplo de relación independiente del contexto, que existe cuando dos hipersecciones comparten unidades. Las relaciones de integración ejemplifican las dependientes del contexto, pues la relación sólo tiene sentido en un contexto determinado y no es inherente a la definición de los componentes. Otros tipos de relaciones son las generalizaciones, inclusiones y exclusiones. Los hiperespacios permiten la representación de ambos tipos de relaciones y su uso en el análisis e integración.

Una propuesta para las reglas de composición, basada en la programación orientada a sujetos [5], establece que estas reglas han de ser una combinación de una regla concisa y general y una serie de reglas específicas y más detalladas. Las últimas se usan para indicar de manera explícita aquellas excepciones a la regla general o para tratar casos especiales no contemplados en la misma. La aplicación de la regla general puede ser un proceso automatizado, pues se aplica a todo o parte de la composición. Los casos especiales que tratan las reglas específicas serán de los tipos siguientes:

- Emparejar unidades con diferentes nombres que describan el mismo concepto.
- Desemparejar unidades con el mismo nombre que describan conceptos diferentes.
- Mediar entre estructuras de módulos diferentes (como establecer correspondencias entre unidades anidadas a diferentes profundidades en diferentes hipersecciones).

Como se ha explicado, en Hyper/J primero se especifica una estrategia de integración general y después se definen las excepciones, o especializaciones, para casos concretos. Actualmente Hyper/J admite tres estrategias generales ([2]) llamadas:

- **mergeByName** indica que las unidades con el mismo nombre en diferentes hipersecciones se corresponden y que se fusionarán en una nueva unidad.
- **nonCorrespondingMerge** indica que las unidades con el mismo nombre en diferentes hipersecciones no se corresponden y por tanto no se fusionarán. Se utiliza cuando, de manera casual, esto ocurre y sin embargo las unidades no están relacionadas.
- **overrideByName** indica que las unidades con el mismo nombre se corresponden y, en lugar de fusionarse, la última de ellas, sobrescribe a las demás. El orden se define según el orden en el que se hayan declarado las hipersecciones en el hipermódulo (prevalece el último).

La sección de relaciones de cualquier hipermódulo empieza con alguna de estas tres estrategias. Si no fuera suficiente sólo con una de estas reglas, a continuación se detallan las reglas específicas. Las posibilidades que tenemos son las siguientes:

Equate:

```
equateRelationship ::= equate unitKind
unitName [, unitName]* [into newName];
unitKind ::= class | interface | operation |
action | field
```

La relación `equate` indica que los elementos de un conjunto de unidades se corresponderán aún cuando sus nombres no sean los mismos. El campo opcional `into` indica el nombre que se le dará a la nueva unidad compuesta. Se puede aplicar a cualquier clase de unidades, siempre y cuando aquellas que participen de la misma declaración sean del mismo tipo.

Order:

```
orderRelationship ::=
   order unitKind unitName [, unitName]*
(before | after)
   unitKind unitName [, unitName]*;
```

```
unitKind ::= hyperslice | class | interface |
operation | action
```

Cuando se combinan métodos, Hyper/J puede, por defecto, elegir el orden en el que ejecutar los métodos originales. O bien, si el orden es importante, con `order` se puede indicar cómo debe hacerse. Nótese que sólo se definen órdenes parciales, es decir, sólo para aquellas unidades en las que sea importante.

Rename:

```
renameRelationship ::=
  rename unitKind unitName to newUnitName;
unitKind ::= class | interface | operation |
action | field
```

Más que una relación, es una directiva de Hyper/J. Permite cambiar el nombre a una unidad concreta de una hipersección de un hipermódulo. Sólo puede cambiarse el nombre en las hipersecciones de salida, nunca en las de entrada (lo cual tampoco es necesario).

Merge:

```
mergeRelationship ::=
 merge unitKind unitName [, unitName]*
 [into newName];
unitKind ::= class | interface | operation |
action | field
```

Combina el conjunto de unidades especificadas, independientemente de si dichas unidades se correspondan o no según la regla general de combinación. Difiere de `equate` en que ésta no implica que las unidades se combinen, si no que sólo indica que se corresponden, dependiendo de la estrategia general el hecho de que se combinen o no.

NoMerge:

```
noMergeRelationship ::=
  noMerge unitKind unitName [, unitName]* ;
unitKind ::= class | interface | operation |
action | field
```

`NoMerge` tiene el efecto opuesto a `merge` (o `override`). Hace que unidades que se corresponden, no se combinen (o sobrescriban), a pesar de lo que implique la regla general de composición. Se utiliza para los casos excepcionales en los que la regla general no debe aplicarse.

Override:

```
overrideRelationship ::=
  override unitKind unitName [, unitName]*
  with unitKind unitName;
unitKind ::= class | interface | operation |
action
```

Esta relación indica que una unidad sobrescribe al conjunto de unidades indicadas en la declaración, en el sentido descrito para `overrideByName`. Puede aplicarse a cualquier tipo de unidad.

Match:

```
matchRelationship ::=
  match unitKind unitName with "pattern";
```

Se usa para indicar que una unidad debe corresponderse con un conjunto de unidades, que se especifican usando un patrón que hace referencia a los nombres de éstas. Al igual que ocurre con `equate`, con `match` las unidades que haga que se correspondan quedarán sujetas a la regla general de composición. Nótese, que la correspondencia sólo se hace entre unidades del mismo tipo. La sintaxis para los patrones se ilustra con los ejemplos siguientes:

`"foo"` indica solo aquellas unidades de nombre `foo`.
`"foo*"` unidades cuyo nombre comience con `foo`.
`"foo*bar"` unidades cuyo nombre comience con `foo` y termine con `bar`.
`"~foo*"` unidades cuyo nombre no comience con `foo`.
`"~foo"` cualquier unidad excepto aquellas de nombre `foo`.
`"{foo,bar}*"` unidades cuyo nombre comience con `foo` o `bar`.
`"*{foo,bar}"` unidades cuyo nombre termine con `foo` o `bar`.
"{f,~foo}*{~r}" unidades cuyo nombre comience con f, pero no con foo y que no termine con la letra r.

Bracket:

```
bracketRelationship ::=
   bracket [classMatchPattern .]
operationMatchPattern
      [from unitKind unitName [, unitName]* ]
      [before fullyQualifiedMethodName,]
      [after fullyQualifiedMethodName,]
unitKind ::= hyperslice | class | operation |
action
```

Esta relación indica que el comportamiento del conjunto de métodos indicados después de la palabra clave `bracket` debe ser ampliado ejecutando antes o después de su invocación los métodos especificados a continuación de `before` y `after`, respectivamente. Opcionalmente se puede restringir con la declaración `from` el contexto donde esta relación tiene efecto. Así puede restringirse a unidades del tipo `action`, `operation`, `class` o `hyperslice`. Esta relación sólo es aplicable a operaciones. En la actual versión de Hyper/J la única información contextual que puede pasarse a los métodos que se ejecutan antes o después es el nombre de la clase o de la operación ampliada. Para ello, se indica con `$ClassName` y `$OperationName`, entre paréntesis, seguido del nombre de las operaciones a invocar.

Summary function:

```
summaryFunctionRelationship ::=
  set summary function for unitType unitName
  to summaryFunction;
summaryFunction ::=
  external unitName
         | unitType unitName
unitType ::= action | operation
```

Cuando se componen métodos que devuelven valores, el método resultante debe devolver un único valor. Por defecto, Hyper/J devuelve el valor devuelto por el último de los métodos según el orden en el que los ha compuesto. En el caso de que esto no sea lo deseado, Hyper/J permite definir una función que sintetice el valor devuelto por cada una de los métodos compuestos. Esta toma como parámetro, la secuencia de valores devueltos y los usa para generar un único valor, que será el que se devuelva. Usando la palabra clave `external`, pueden usarse funciones que no estén en el hiperespacio definido. Nótese que este tipo de funciones han de ser siempre estáticas.

2.6 LIMITACIONES DE HYPER/J

Con Hyper/J se puede descomponer un programa de acuerdo con diferentes competencias, creando nuevos módulos separados que no interfieran unos con los otros. Posteriormente pueden integrarse para obtener un resultado final o de cualquier fase intermedia. Hyper/J ayuda a definir las interacciones entre las diferentes descomposiciones, proveyendo de una capacidad de composición. Por ejemplo, se puede usar para crear una versión de un sistema software que contenga solamente determinadas características y no otras, incluso cuando el sistema original no fue diseñado con dichas características de manera separada; o puede usarse para extender o adaptar un componente, incluso cuando dicho componente no se ha escrito facilitando los puntos de enlace necesarios.

Hyper/J puede usarse en cualquier fase del ciclo de vida del software, ya sea diseño, implementación, integración, evolución del sistema y reingeniería. Cuando se usa durante el diseño o la implementación, Hyper/J permite separar todos las competencias de importancia desde el principio. Durante la integración del sistema, los mecanismos de integración de Hyper/J pueden usarse para integrar el software que se ha desarrollado de manera separada, incluyendo los componentes reusables. Usado durante la evolución del sistema, los mecanismos de separación de competencias de Hyper/J permiten a los programadores centrarse únicamente en aquellas piezas del sistema relevantes en cada momento, no siendo necesario modificar el código existente. En la fase de reingeniería, se puede utilizar Hyper/J para introducir nuevas descomposiciones sin necesidad de cambios en el código actual.

Referente a las unidades de las que hemos estado hablando, Hyper/J soporta como unidades los paquetes, interfaces, clases, funciones y atributos Java. Soporta una matriz de competencias con esas unidades y permite definir hipersecciones con conjuntos de unidades para después integrarlos en hipermódulos. Hyper/J genera ficheros Java class para todos los hipermódulos definidos, que pueden ser ejecutados, si lo fueren, o usados como piezas de otras aplicaciones.

Las principales limitaciones actuales en la funcionalidad de la herramienta de composición de Hyper/J están detalladas en la sección 4.5 del manual de usuario e instalación de Hyper/J [2] y, por tanto, no se repetirán aquí.

3. EJEMPLO

Para poner en práctica los anteriores conceptos teóricos de Hyper/J se trabajará con un sistema de ejemplo para la gestión de personal. Partimos de un sistema orientado a objetos con dos funcionalidades principales: la gestión de la información del personal (nombre, identificador y cargo en la empresa) y la gestión de lo referente a los salarios. Además, gracias a un método `check()`, se cumple que el número de jefes por empleado es como mucho de 3. Originalmente tendríamos la siguiente jerarquía de clases:

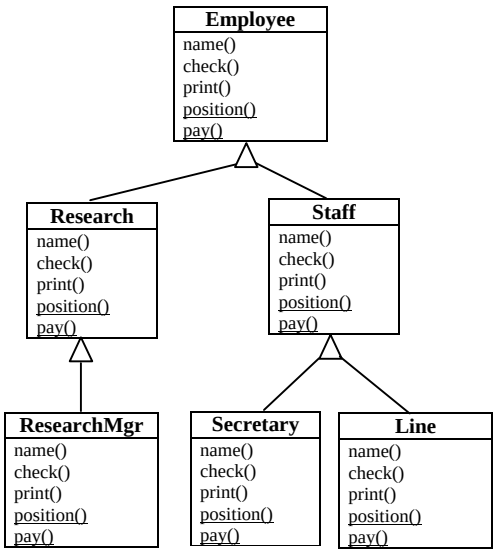


Figura 1 Jerarquía de clases del sistema

Supongamos que en este contexto nos aparecen nuevos requisitos. Por un lado, queremos eliminar la gestión de los salarios del sistema de gestión del personal y, por otro lado, modificar la actual regla referente a los jefes, pasando éstos a ser como mucho uno sólo por empleado. Gracias a Hyper/J podremos llevar a cabo estos cambios sin alterar el actual sistema de modularización y manteniendo las actuales relaciones entre las clases. En Hyper/J tendríamos la siguiente definición:

```
package Personnel:   Feature.Personnel
operation position:  Feature.Payroll
operation pay:       Feature.Payroll
```

Con la anterior definición, por defecto, tendremos que todos los miembros de las clases e interfaces del paquete Java `Personnel` pertenecen a la competencia `Personnel` en la dimensión `Feature`. Por sobre-

escritura, las declaraciones `operation` indican que cualquier método en cualquier clase que se llame `position()` o `pay()` pertenece a la competencia `Payroll` en la dimensión `Feature`. Gráficamente podemos visualizar la matriz de competencias como lo muestra figura 2:

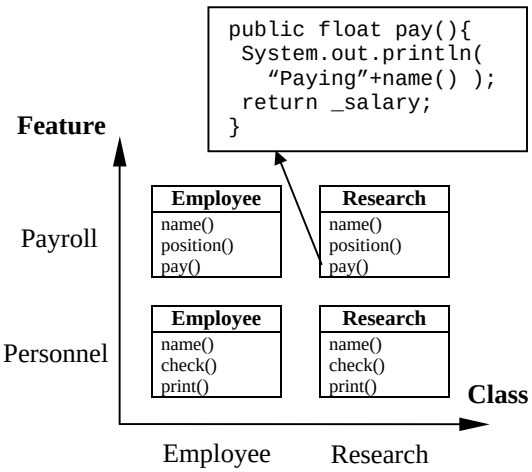


Figura 2 Matriz de comptencias con las dimensiones Feature y Class

En la figura pueden verse las dos dimensiones `Class` y `Feature` y dos hipersecciones, `Personnel` y `Payroll`. Cada hipersección encapsula su propia estructura de clases. Tal como se ha definido, `Payroll` contiene los métodos `position()` y `pay()`, además del método abstracto `name()` ya que éste es invocado desde `pay()`. Como se ha explicado en el apartado Hyperspaces cada hipersección debe ser declarativamente completa y es por ello que es necesario incluir el método abstracto `name()`, pues es referenciado en el método `pay()`. En la otra hipersección, `Personnel`, las clases contienen todos los otros métodos originales. Hyper/J inserta automáticamente una declaración abstracta para cualquier miembro al que se haga referencia, aunque no se de su implementación. De esta forma, las hipersecciones están desacopladas y pueden ser manejadas de manera independiente. `Personnel` es un programa completo, mientras que `Payroll` es un componente Java correcto, a falta de indicar una implementación para el método abstracto `name()`.

A continuación, para componer las hipersecciones usando Hyper/J es necesario crear un hipermódulo que indique las hipersecciones que se desea componer y cómo se llevará a cabo la composición. El siguiente fragmento de código define un hipermódulo con las hipersecciones requeridas:

```
hypermodule PayrollPlusPersonnel
   hyperslices: Payroll, Personnel;
   relationships:
        mergeByName;
end hypermodule
```

Como puede verse, primero se indica el nombre del hipermódulo, luego la lista de hipersecciones a componer y luego el tipo de composición que se llevará a cabo, que en este caso será por nombre. Ello implica que las entidades que en cualquiera de las hipersecciones tengan el mismo nombre, serán las mismas y habrán de fusionarse en una sola. Por ejemplo, `Payroll.Employee` y `Personnel.Employee` tienen el mismo nombre y se combinan para dar una nueva clase en `PayrollPlusPersonnel`. Los métodos concretos en la hipersección `Personnel` implementan los métodos abstractos de `Payroll`.

Otras competencias podrían modularizarse de modo separado. Puede ser el caso de querer establecer ciertas reglas de negocio, como la que se dijo al principio de restringir el número de jefes de un empleado a uno. Así por ejemplo, tendríamos las siguientes declaraciones para el método `check()` de `Employee`:

```
method Personnel.Employee.check:
BusinessRule.ThreeManagers

method SingleManagerPkg.Employee.check:
BusinessRule.OneManager
```

Así aparecería en la matriz de competencias una nueva dimensión `BusinesRule`, como se ve en la figura 3:

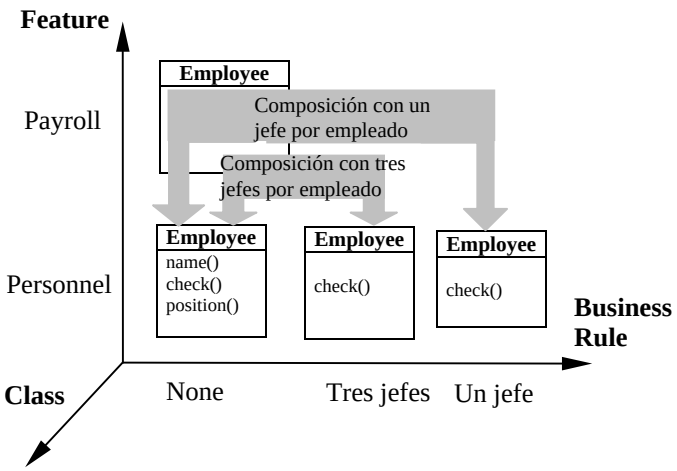


Figura 3 Nueva matriz de competencias

Como se observa en el gráfico, es necesaria la definición de la competencia `None` en las reglas de negocio, de lo cual se encarga automáticamente Hyper/J. `None` contiene todo el código que no tiene nada que ver con las reglas de negocio, pero que, como se explicó en el apartado referente a la matriz de competencias, es necesario incluir. Hyper/J automáticamente inserta una declaración abstracta para el método `check()`. Como resultado el desarrollador puede combinar las reglas de negocio en hipermódulos bien usando `BusinessRule.OneManager` o `BusinessRule.ThreeManagers`.

4. CONCLUSIONES

La filosofía de Hyper/J para soportar la separación de competencias es permitir la integración de módulos, denominados en la terminología *hyperslices*. Cada uno de estos módulos encapsula una competencia específica y se programa en Java estándar, por lo que no se requieren extensiones ni constructores nuevos para el lenguaje. Los *hyperslices* pueden solaparse o existir de manera independiente unos de los otros. Para hacer posible la integración, existen reglas de composición de modo que, en principio se sigue una regla general establecida de composición y después se utilizan reglas específicas, que se refieren a casos particulares.

En Hyper/J, a diferencia de otras implementaciones de metodologías de separación de conceptos, no existe el concepto de un programa base, sino que cada componente de código es independiente y provee de forma completa el código para la unidad particular de descomposición. Esa es la principal ventaja de Hyper/J, y es el hecho de utilizar componentes verdaderamente independientes entre sí, lo cual beneficia la reusabilidad de los mismos (llegando al caso de aplicaciones que se diseñaron por separado y que son capaces de funcionar independientemente).

Una ventaja de Hyper/J es el hecho de trabajar con el bytecode, por lo que no se necesita el código fuente, sino solamente conocer la interfaz (nombres de las clases, funciones...), lo cual se puede obtener fácilmente con compiladores inversos o con la documentación. Sin embargo, la herramienta de la que se dispone está aún en desarrollo, por lo que tiene algunas limitaciones.

El uso de Hyper/J es conveniente cuando se trata de combinar grandes componentes con muchas clases. Para combinar distintos componentes y probar diferentes configuraciones, en lugar de tener que trabajar añadiendo o quitando cada una de las clases, podemos beneficiarnos del uso de las hipersecciones. Éstas agrupan todas las clases y métodos asociados con una competencia específica, por lo que podemos trabajar con éstas combinándolas o eliminándolas según las necesidades, implicando una sola acción, en lugar de tener que manejar muchos módulos pequeños.

5. REFERENCIAS

1. Aspect Oriented Software Development. http://aosd.net.

2. Peri Tarr, Harold Ossher. *Hyper/J™ User and Installation Manual*, IBM Research, 2000.

3. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. *N degrees of separation: Multi-dimensional separation of concerns*. In Proceedings of the 21st ICSE'99.

4. D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, December 1972.

5. H. Ossher, M. Kaplan, W. Harrison, A. Katz, V. Kruskal, *Subject-oriented composition rules*, OOPSLA'95, ACM, October 1995, pp. 235-250.

6. Barry R. Pekilis. Multidimensional Separation of Concerns and IBM Hyper/J. January 22, 2002.

AspectJ en la Programación Orientada a Aspectos

Juan Manuel Nieto Moreno [*]
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT

La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables.

AspectJ es una extensión al lenguaje Java orientada a aspectos y de propósito general, que extiende el popular lenguaje con una nueva clase de módulos llamados aspectos. Los aspectos cortan las clases, las interfaces y a otros aspectos mejorando la separación de competencias y haciendo posible localizar de forma limpia los conceptos de diseño del corte.

[*] Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

1 INTRODUCCIÓN AL LENGUAJE

En AspectJ, un aspecto es una clase, exactamente igual que las clases Java, pero con una particularidad, que pueden contener unos constructores de corte, que no existen en Java. Los cortes de AspectJ capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, de constructores, lanzamiento de excepciones, acceso a atributos... Los veremos con más detalles en apartados siguientes. Los cortes no definen acciones, sino que describen eventos. De modo que en AspectJ los aspectos son constructores que trabajan cortando de forma transversal la modularidad de las clases de forma clara y específica.

Cualquier programa Java que sea válido, es también un programa valido en AspectJ. El compilador de AspectJ genera ficheros class conformes a la especificación Java byte-code, por lo que cualquier máquina virtual Java

(JVM) debe poder ejecutarlos. Al usar Java se obtienen todos los beneficios de este lenguaje, los cuales han quedado demostrados por su extensa aceptación entre la comunidad de programadores. Ello acelera además el proceso de aprendizaje en el lenguaje, puesto que sólo se ha de aprender a utilizar la parte nueva introducida por AspectJ.

Debemos diferenciar en AspectJ dos partes: la especificación del lenguaje y su implementación. La primera define el lenguaje en el que se escribe el código; en AspectJ la funcionalidad de los conceptos encapsulados se escribe en Java estándar, como el que ya todos conocemos y las extensiones del lenguaje se usan para definir dónde y cómo han de entrelazarse con el código de nuestra aplicación. La segunda parte es la implementación del lenguaje, provista de herramientas para compilar, depurar e integrar AspectJ con conocidos entornos de desarrollo integrados (*integrated development environment*, IDE) como JBuilder, Eclipse, Emacs...

Existen una serie de conceptos que son necesarios entender y conocer antes de poder utilizar AspectJ y que no tienen por qué entenderse de igual manera en los diferentes lenguajes que implementan los conceptos de la programación orientada a aspectos. Se hará referencia siempre al término inglés utilizado para designar estos conceptos en la documentación del lenguaje para facilitar así la lectura de otros documentos. No obstante al principio se da una explicación del significado de estos.

2 CROSSCUTTING EN ASPECTJ

En AspectJ, la implementación de las reglas de entrelazado se denomina *crosscutting* y a los conceptos encapsulados que originalmente estaban repartidos por todo o parte de la aplicación se les denomina *crosscutting concerns*. El proceso de combinación de éste código con el de la aplicación Java se conoce como *weaving process*, es decir, proceso de entrelazado. Las reglas de entrelazado definen de una manera sistemática las diferentes partes del código que deben ser afectadas por los conceptos encapsulados en los aspectos, respondiendo a la cuestión qué hacer cuando cierto punto de la ejecución de un programa se alcanza. A dichos puntos de ejecución se les llama *pointcuts*. En AspectJ se definen dos tipos de *crosscutting*: estático y dinámico.

2.1 CROSSCUTTING DINÁMICO

Se dice que un aspecto afecta a la aplicación de manera dinámica, o usando el término inglés, se habla de *dynamic crosscutting*, cuando dicho aspecto afecta al comportamiento global o una parte de la aplicación. *Crosscutting* dinámico es la forma más frecuente en AspectJ de entrelazado y es la manera de dotar de nuevo comportamiento a la ejecución de un programa. De esta forma se puede enriquecer o según sea el caso, reemplazar el flujo de ejecución de un programa, de forma que éste vaya por diferentes requisitos que en nuestro código se tengan encapsulados en diferentes módulos. Por ejemplo, algo muy frecuente es indicar que cierta acción ha de ser realizada antes de la ejecución de ciertos métodos o de la captura de determinadas excepciones dentro de un conjunto de clases, definiendo en un módulo aparte los puntos dónde el código se ha de entremezclar (*weaving points*) y la acción a realizar en el momento en el que se alcancen dichos puntos.

2.2 CROSSCUTTING ESTÁTICO

Se dice que un aspecto afecta a la aplicación de manera estática, o usando el término inglés, se habla de *static crosscutting*, cuando se añaden modificaciones en la estructura estática del programa, esto es, en las clases, interfaces y aspectos. Por tanto, este tipo de técnica no implica modificación del comportamiento del sistema. En la mayor parte de los casos su uso se requiere para dar soporte a la implementación de los conceptos de *crosscutting* dinámico. Por ejemplo, puede simplificarse la implementación de éstos modificando las relaciones de herencia entre clases e interfaces, para poder trabajar así con las clases abstractas en lugar de tener que hacerlo con las implementaciones específicas; o puede ser de utilidad modificar los atributos o métodos de una clase o interfaz para poder así definir estados específicos y comportamientos que pueden ser usados en las acciones que se lleven a cabo con entrelazado dinámico. También es posible con entrelazado estático definir advertencias (*warnings*) y errores (*errors*) en tiempo de compilación de una manera sencilla y que afecte a todo o una parte de la aplicación, permitiendo definir los puntos de corte de una manera general con predicados lógicos.

3 LAS REGLAS DE ENTRELAZADO

AspectJ es una extensión al lenguaje de programación Java para especificar las reglas de entrelazado para los aspectos que afectan a una aplicación ya sea de forma estática o dinámica. Las extensiones están diseñadas de forma que el proceso de aprendizaje de un programador Java para usarlas sea mínimo. A continuación se explican los constructores que utiliza AspectJ para especificar las reglas de entrelazado de los aspectos con la aplicación de manera precisa (al igual que en casos anteriores, especifico en primer lugar el término inglés):

- **Join points**, puntos de enlace. Un punto de enlace es un sitio identificable en la ejecución de un programa. Podría ser una llamada a un método o la asignación de un nuevo valor a un atributo de una clase. En AspectJ todo gira alrededor de los puntos de enlace, ya que son los lugares donde los valores añadidos de los aspectos son enlazados con el código.

- ***Pointcuts***, puntos de corte. Estos son constructores donde se especifican los puntos de enlace y se captura información referente al contexto. Por ejemplo, un punto de corte puede referirse a un punto de enlace que sea una llamada a un método, pudiendo, en tal caso, capturar el contexto del método, siendo éste, el objeto sobre el que se invoca el método, así como los argumentos de la llamada. Para entender la diferencia entre puntos de enlace y puntos de corte, se puede pensar en estos últimos como las reglas de especificación y en los primeros como las situaciones que satisfacen dichas reglas.

- ***Advice***, avisos. Este es el código que se debe ejecutar en los puntos de enlace que se especifiquen en un punto de corte. Los avisos se pueden ejecutar antes, después o alrededor. Alrededor quiere decir que se puede, con el aviso, modificar la ejecución del código a la altura del punto de enlace, reemplazarlo o ignorarlo, si se es eso lo que se desea. Por ejemplo, se podría especificar un aviso para registrar en un fichero histórico cierto mensaje, siempre antes de la ejecución de ciertas funciones que se encuentran dispersas por diferentes módulos. El cuerpo de un aviso se parece mucho al cuerpo de un método, ya que encapsula la lógica que debe ser ejecutada cuando se alcanza cierto punto de enlace en la ejecución. Los puntos de corte junto con los avisos, son las herramientas para implementar entrelazado dinámico. Los puntos de corte identifican dónde y los avisos lo completan indicando qué hacer.

- ***Introduction***, introducciones. Si lo anterior era para el entrelazado dinámico, las introducciones se usan para el estático. Con ella es posible introducir cambios a las clases, interfaces y aspectos del sistema para así permitir el entrelazado dinámico. Conlleva cambios estáticos en los módulos que no afectan directamente a su comportamiento. Por ejemplo, se pueden añadir métodos o atributos a clases y después usarlos tal como si hubieran sido definidos por la propia clase.

- ***Compile-Time declaration***, declaraciones en tiempo de ejecución. Esta es otra forma de implementar técnicas de entrelazado estático. Permite añadir advertencias y errores para en tiempo de compilación detectar ciertos patrones de uso que queramos advertir o prohibir (en el caso de que los identifiquemos

como errores). Por ejemplo, se puede declarar que es un error hacer uso de las llamadas del paquete `java.sql` fuera del paquete `dataAccess` de la aplicación.

– ***Aspect***, aspectos. Estos son la unidad central de AspectJ, al igual que lo son las clases en Java. Contienen el código que expresa las reglas de entrelazado tanto para los aspectos dinámicos como los estáticos. Puntos de corte, avisos, introducciones y declaraciones de compilación se combinan en los aspectos. Además de estos elementos propios de AspectJ, los aspectos pueden contener atributos, métodos y clases anidadas, tal como los tienen las clases normales en Java.

Veamos cómo funciona todo ello junto. Cuando se diseña un comportamiento que se encuentra disperso por la aplicación, lo primero que hay que hacer es identificar los puntos de enlace donde se quiere enriquecer el comportamiento o bien modificarlo. Una vez hecho, se diseña cual será el nuevo comportamiento. Para hacer esto, primero se escribe un aspecto que sirva como módulo para el comportamiento global. En él se escriben los puntos de corte para capturar los puntos de enlace que se deseen. Finalmente, se crean avisos para esos puntos de corte y se define dentro del cuerpo de los avisos las acciones que se deben realizar cuando se ejecute el código que definen los puntos de enlace. Es posible que para permitir ciertos avisos sea necesario hacer uso de entrelazado estático.

Un ejemplo sencillo sería el siguiente: considere una aplicación en la que queremos añadir un aspecto para registrar la ejecución de todos los métodos públicos de dete
rminados paquetes. Lo primero es crear el aspecto que encapsule dicho comportamiento, que como es fácil observar, está distribuido por diferentes clases. Después se escribe un punto de corte en el aspecto para capturar todos los puntos de enlace para las operaciones públicas en el conjunto de clases que se desee. Por último, se escribe un aviso para este punto de corte, donde, en su cuerpo, se escribirían las instrucciones para registrar la llamada en un fichero o de cualquier otra forma. Si se quisiera mantener algún tipo de estado sobre las clases sobre las que se han registrado las llamadas, como podría ser el número de métodos ejecutados en cada clase, debería usarse introducción estática en el aspecto para añadir un atributo entero a todas las mismas clases anteriores. En tal caso, el aviso puede actualizar y leer este entero y escribirlo también a la información que se registra. El siguiente apartado nos muestra un ejemplo muy parecido.

4 DETALLES DEL LENGUAJE

En esta sección se presentan los aspectos más relevantes del lenguaje AspectJ, tratando en primer lugar los puntos de corte, viendo su sintaxis y modo de uso; los avisos, sintaxis y tipos; el manejo de la información contextual; las reglas de entrelazado estático disponibles; extensión e instancias de los aspectos; y por último, una visión rápida de la API de AspectJ.

4.1 PUNTOS DE CORTE

Los puntos de corte (*pointcuts*) capturan o identifican puntos de enlace en el flujo del programa. Una vez definidos con los constructores que proporciona el lenguaje, puede especificarse mediante reglas de entrelazado cómo combinar los comportamientos definidos en los aspectos con el código original. Además los puntos de corte dan acceso al contexto del programa y las acciones se pueden beneficiar de la información contextual para llevar a cabo su cometido.

En AspectJ las *pointcuts* se pueden declarar anónimos o con un nombre. Al igual que las clases, un punto de corte anónimo se define en el lugar donde se usa, que bien podría ser en un *advice* o en la definición de otro *pointcut*. Los puntos de corte que se definen con un nombre pueden después ser referenciados desde múltiples partes del código, haciéndoles reusables. Su sintaxis es la siguiente:

```
pointcut nombre(argumentos) : definición
```

Por `definición` se entiende el conjunto de reglas que definen los puntos de enlace donde se quiere insertar cierto comportamiento. El nombre que se utilice para definir el *pointcut* es el que luego ha de utilizarse en los *advices* para hacer referencia a ellos. Véase el siguiente ejemplo:

```
pointcut operacionesPublicas(): call(public *.*(..));

before() : operacionesPublicas(){
    //acciones a realizar antes de las llamadas a las funciones públicas
}
```

Es posible también definir puntos de corte anónimos, es decir, sin un nombre que los referencie. Se definen en el lugar donde se usan y no pueden reusarse. No se aconseja, por tanto, usarlos si la expresión que los define es complicada o interesa utilizarlos en varios *advices*. Su sintaxis dentro de la definición de los *advices* es la siguiente:

```
[especificación-del-advice]: definición-del-pointcut
```

Si quisiéramos escribir el mismo ejemplo anterior pero con el *pointcut* anónimo, se haría como lo muestra el siguiente ejemplo –obsérvese que lo importante, la definición del punto de enlace, es igual al anterior:

```
before(): call(public *.*(..)) {
     //acciones a realizar antes de las llamadas a las funciones públicas
}
```

SINTAXIS Y SEMÁNTICA DE LOS PUNTOS DE ENLACE

Los puntos de corte harán referencia a una serie de puntos de interés (*joinpoints*) en el programa, por lo que se necesita un lenguaje eficiente para definirlos. AspectJ utiliza una sintaxis basada en elementos comodines que sirven para capturar puntos de enlace que comparten características comunes. Dichos elementos son los siguientes:

- * un asterisco denota cualquier número de caracteres excepto el punto.
- .. dos puntos seguidos denota cualquier número de caracteres incluyendo el punto.
- + el símbolo de la suma denota los descendientes de una clase.

Más adelante se verá que el significado de estos caracteres dependerá de los elementos a los que acompañen. Al igual que en Java y en muchos lenguajes las expresiones se pueden combinar para dar lugar a expresiones más complejas. Para ello se utilizan los siguientes operadores unarios y binarios:

- `!` la exclamación de cierre es un operador unario que denota todos los puntos de enlace, excepto aquellos que se especifiquen en la expresión que siga. Por ejemplo, `!within(aspects.*)` denota todos los puntos de enlace excepto aquellos del paquete `aspects`.
- `||` dos barras verticales es el operador binario que permite combinar puntos de enlace con nombre y anónimos, de tal forma que se cumpla alguno de los lados de la expresión.
- `&&` es el operador binario que permite combinar puntos de enlace con nombre y anónimos, de tal forma que se cumplan ambos lados de la expresión.

Además pueden usarse los paréntesis combinados con estos operadores con objeto de alterar la precedencia por defecto de las expresiones y facilitar también la legibilidad. Los elementos expuestos tendrán diferente significado según acompañen a paquetes, tipos (clases, interfaces o aspectos), métodos o atributos. Veamos algunos ejemplos del significado que adquieren según cómo y donde se usen:

Tabla 1 Patrones de puntos de enlace

Patrón	**Elementos referidos**
`Manager`	Tipos de nombre `Manager`.
`*Manager`	Tipos con un nombre que termine en `Manager`.
`java.*.Name`	Tipo `Name` en cualquiera de los subpaquetes del paquete `java`.
`java..*`	Cualquier tipo del paquete `java` o todos sus subpaquetes

`javax..*Set+`	Todos los tipos en el paquete `javax` o sus subpaquetes cuyo nombre termine en `Set` y sus subtipos.
`public void Stack.clean()`	El método `clean()` de la clase `Stack` que tenga acceso público, devuelva `void` y no tenga argumentos.
`public void Stack.push(Integer) throws RangeExceded`	El método `push()` de la clase `Stack` que tenga acceso público, devuelva `void`, tenga un único argumento de tipo `Integer` y lance la excepción `RangeExceded`.
`public void Stack.push*(*)`	Todos los métodos públicos de la clase `Stack` cuyo nombre empiece con `push` y tenga un único argumento de cualquier tipo.
`public void Stack.*()`	Todos los métodos en de la clase `Stack` que tengan acceso público devuelvan `void` y no tengan argumentos.
`public * Stack.*()`	Todos los métodos públicos de la clase `Stack` que no tengan argumentos y devuelvan cualquier tipo.
`public * Stack.*(..)`	Todos los métodos públicos de la clase `Stack` que tomen cualquier número de argumentos.
`* Stack.*(..)`	Todos los métodos de la clase `Stack`.
`!public * Stack.*(..)`	Todos los métodos que no tengan acceso público de la clase `Stack`.
`public static void MyClass.main (String[] args)`	El método estático `main()` de la clase `MyClass` con acceso público.
`* Stack+.*(..)`	Todos los métodos de la clase `Stack` o sus subclases.
`* java.io.Reader.read(char[],..)`	Métodos `read()` de la clase `Reader` con cualquier número de argumentos, con el primero del tipo `char[]`.
`* javax..*.add*List(List+)`	Métodos que empiecen con `add` y terminen en `List` en el paquete `javax` o sus subpaquetes y que tomen un argumento del tipo `List` o sus subtipos.
`* *.*(..) throws SQLException`	Cualquier método que lance `SQLException`.
`public MyClass.new()`	Constructores públicos de la clase `MyClass` sin argumentos.
`public MyClass.new(Integer)`	Constructores públicos de la clase `MyClass` que tomen un único argumento de tipo `Integer`.
`public MyClass.new(..)`	Todos los constructores de la clase `MyClass` que tomen cualquier número de argumentos.
`public *MyClass.new(..)`	Constructores públicos de clases cuyo nombre termine en `MyClass`.
`public MyClass+.new(..)`	Constructores públicos de la clase `MyClass` o sus subclases.
`public MyClass(..) throws InvalidFormatException`	Constructores públicos de la clase `MyClass` que lancen InvalidFormatException.
`private float MyClass.value`	Atributo privado `value` de la clase MyClass.
`* MyClass.*`	Todos los atributos de la clase MyClass.
`!public static * MyClass..*.*`	Todos los atributos no públicos estáticos de la clase MyClass y de sus subpaquetes.
`Public !final *.*`	Atributos no finales públicos de cualquier clase.

Como se ve en la tabla, el significado de los caracteres comodines depende si se aplican a tipos, métodos o atributos. Referido a los tipos, el asterisco especifica una parte de la clase, interfaz o paquete; los dos puntos seguidos se usan para denotar todos los subpaquetes; y el símbolo de suma denota los subtipos (subclases o subinterfaces). Mientras que referido a los métodos, los dos puntos seguidos denotan cualquier tipo y número de argumentos que tome el método.

IMPLEMENTANDO LOS PUNTOS DE CORTE

Una vez que tenemos una manera eficiente de definir los puntos de enlace en el código, se necesita una manera de utilizarlos. Para referirnos a ellos podemos hacerlo según el tipo de punto de enlace al que representan, como puede ser llamadas a métodos, ejecución de estos, lectura de atributos... Véase la siguiente tabla para una descripción de las diferentes posibilidades que ofrece AspectJ. En la terminología del lenguaje se hace referencia a ellos como *Kinded pointcuts*.

Tabla 2 Tipos de puntos de corte

Categoría	**Sintaxis**
Ejecución de métodos	`execution(MethodSignature)`
Llamada a métodos	`call(MethodSignature)`
Ejecución de constructores	`execution(ConstructorSignature)`
Llamada a constructores	`call(ConstructorSignature)`
Inicialización de clases	`staticinitialization(TypeSignature)`
Lectura de atributos	`get(FieldSignature)`
Escritura de atributos	`set(FieldSignature)`
Captura de excepciones	`handler(TypeSignature)`
Inicialización de objetos	`initialization(ConstructorSignature)`
Pre-inicialización de objetos	`preinitialization(ConstructorSignature)`
Ejecución de avisos	`adviceexecution()`

Se puede escribir un sencillo ejemplo para comprender la cantidad de puntos de enlace que tenemos en una pequeña aplicación e imaginarnos él número de éstos que podemos llegar a tener cuando se trate de una aplicación real. Definamos el siguiente aspecto:

```
public aspect ImprimeTodos{

  pointcut todos(): if(true);

   before(): todos() && !within(ImprimeTodos){
        System.out.println(thisJoinPoint);
   }
}
```

Para definir el *pointcut* `todos()` se usa un punto de corte condicional (véase Tabla 7) cuya condición lógica está siempre a cierto. Para evitar caer en un bucle infinito se utiliza los llamados puntos de corte basados en localización (véase Tabla 4), de modo que se excluyen los puntos de enlace de la propia clase mediante `!within`. El programa que utilizaremos como ejemplo es el siguiente:

```
public class MiClase{

   public static void main(String args[]){
       System.out.println("* Inicio *");
       new MiClase().operacion();
   }

   public void operacion(){
      System.out.println("* Hola *");
   }
}
```

Ejecutándolo la clase `MyClass` con el aspecto `ImprimeTodos` tenemos la siguiente salida donde se ve cómo se han alcanzado casi todos los `pointcuts` expuestos en la Tabla 2:

```
staticinitialization(MiClase.<clinit>)
execution(void MiClase.main(String[]))
get(PrintStream java.lang.System.out)
call(void java.io.PrintStream.println(String))
* Inicio *
call(MiClase())
initialization(MiClase())
execution(MiClase.<init>)
execution(MiClase())
call(void MiClase.operacion())
execution(void MiClase.operacion())
get(PrintStream java.lang.System.out)
call(void java.io.PrintStream.println(String))
* Hola *
```

Este ejemplo nos muestra la cantidad de posibilidades que tenemos a la hora de definir puntos de enlace. Para poder ser más concretos en la definición de los puntos de corte nos podemos servir de un buen número de *pointcuts* disponibles en AspectJ y que se muestran en las siguientes tablas.

El siguiente conjunto de *pointcuts* captura puntos de enlace siempre y cuando ocurran en el contexto de otro *pointcut*. Por eso, declaran siempre otro punto de corte como argumento. Por ejemplo, `cflow(doIt())`, identifica los puntos de enlace que ocurren entre recibir las llamadas al método `doIt` y volver de estas llamadas (incluyendo la propia llamada a `doIt`). Además de `cflow(Pointcut)` existe otra posibilidad, `cflowbelow(Pointcut)`, cuya utilidad veremos con un ejemplo a continuación de la siguiente tabla:

Tabla 3 Pointcuts basados en control-flow

Pointcut	**Descripción**
`cflow(call(* MyClass.operation(..))`	Puntos de enlace durante la ejecución del método `operation()` de la clase `MyClass`, incluyendo la llamada al método `operation()` de `MyClass`.
`cflowbelow(` `  call(* MyClass.operation(..))`	Puntos de enlace durante la ejecución del método `operation()` de la clase `MyClass`, excluyendo la llamada al método `operation()` de `MyClass`.
`cflow(Operations())`	Todos los puntos de enlace en el contexto de los puntos de enlace capturados por el punto de corte `Operations()`.
`cflow(staticinitializer(StClass))`	Puntos de enlace durante la inicialización de la clase `StClass`.
`cflowbelow(execution(MyClass.new(..))`	Puntos de enlace durante la ejecución de alguno de los constructores de la clase `MyClass`, excluyendo la ejecución misma del constructor.

El *pointcut* `cflowbelow()` puede ser de mucha utilidad en los algoritmos recursivos. La recursividad es una técnica muy útil, pero que genera muchas llamadas y, consecuentemente, muchos puntos de enlace repetitivos. Por lo general, los más interesantes de ellos serán los referentes a la primera y la última llamada. El siguiente ejemplo muestra como `cflowbelow()` puede ser de utilidad en este caso. Sea el siguiente programa recursivo que muestra la jerarquía de clases de un objeto:

```
public class Jerarquia{
   public static void main(String args[]){
        try{
               padres(Class.forName(args[0]));
        }catch(Exception e){}
   }

   static void padres(Class obj){
      if (null==obj)
         return;
       System.out.println(obj.getName());
       padres(obj.getSuperclass());
   }
}
```

Si probamos la aplicación `Jerarquia` con la clase `javax.servlet.http.HttpServlet` como entrada, obtendremos la siguiente salida:

```
javax.servlet.http.HttpServlet
javax.servlet.GenericServlet
java.lang.Object
```

En el caso de que quisiéramos enlazar avisos con las llamadas recursivas, podría interesarnos distinguir únicamente la primera llamada. El siguiente aspecto define dos *pointcuts*: `llamadas()`, que captura todas las llamadas al método `padres(Class)`; y `primera()` que captura sólo la primera de dichas llamadas.

```
public aspect LlamadasRecursivas{

  pointcut llamadas(): call(void Jerarquia.padres(Class));
  pointcut primera(): llamadas() && !cflowbelow(llamadas());

  before(): llamadas() {
       System.out.println("Otra: "+thisJoinPoint);
  }
```

```
  before(): primera() {
       System.out.println("Primera: "+thisJoinPoint);
  }

}
```

Cómo es deseado, el segundo de los avisos sólo se ejecutará una vez –antes de la primera llamada a `padres(Class)`. Combinando la clase `Jerarquia` con el aspecto y dándole la misma entrada tenemos la siguiente salida:

```
Otra: call(void Jerarquia.padres(Class))
Primera: call(void Jerarquia.padres(Class))
javax.servlet.http.HttpServlet
Otra: call(void Jerarquia.padres(Class))
javax.servlet.GenericServlet
Otra: call(void Jerarquia.padres(Class))
java.lang.Object
Otra: call(void Jerarquia.padres(Class))
```

El siguiente grupo de puntos de corte que se definen son los que se basan en la localización del código referenciado. Hay dos categorías: `within(TypePattern)` y `withincode(MethodSignature)`. El primero se usa para capturar todos los puntos de enlace en el cuerpo de las clases, aspectos o subclases especificados en `TypePattern`. El segundo se usa para capturar los puntos de enlace dentro de una estructura léxica de una clase o un método. La siguiente tabla muestra algunos ejemplos de su uso:

Tabla 4 Pointcuts basados en localización

Pointcut	**Descripción**
`within(MyClass)`	Puntos de enlace dentro de la clase `MyClass`.
`within(MyClass+)`	Puntos de enlace dentro de la clase `MyClass` y sus subclases.
`withincode(* MyClass.operation(..))`	Puntos de enlace dentro del método `operation()` de la clase `MyClass`.
`withincode(* *printer.flow(..))`	Puntos de enlace dentro del método `flow()` en clases cuyo nombre acabe en `printer`.

Un uso común de `within()` es para excluir los puntos de enlace del aspecto donde se define. Por ejemplo, el siguiente punto de corte excluye los puntos de enlace correspondientes a las llamadas a los métodos `print` de la clase `java.io.Stream` que ocurran dentro del aspecto `TraceAspect`:

```
call(* java.io.PrintStream.print*(..)) && !within(TraceAspect)
```

Los siguientes puntos de corte, `this` y `target`, están basados en los tipos de los objetos en tiempo de ejecución. Con `this` se referencia al objeto actual, mientras que con `target` es al objeto sobre el que se llama al método. Estos puntos de corte sirven además para recoger el contexto en el punto de enlace. `this` toma la forma `this(Type)` y así captura los puntos de enlace asociados con objetos del tipo especificado `Type`. Al igual, `target` es también de la forma `target(Type)`.

Tabla 5 Pointcuts de objetos

Pointcut	**Descripción**
`this(MyClass)`	Puntos de enlace donde `this` es una instancia de `MyClass` o sus subclases (`this` es el objeto que se esté ejecutando actualmente).
`target(MyClass)`	Puntos de enlace en los que el objeto donde se llama el método es instancia de `MyClass` o sus subclases.

Nótese como `this()` y `target()` toman como argumentos `Type` y no `TypePattern`. Eso quiere decir que no podremos usar los comodines asterisco o dos puntos seguidos (el comodín referente a los subtipos, `+`, no se necesita, pues éstos son capturados por las reglas de herencia de Java). Obsérvese también que, al no ejecutarse

los métodos estáticos sobre un objeto, éstos no tendrán un objeto `this` asociado, por lo que el punto de corte no incluirá este tipo de métodos (igualmente ocurre con `target)`. Esta última es la diferencia de uso que puede hacerse con `within`.

El siguiente *pointcut* que se basa en el tipo de los argumentos del punto de enlace. Por argumentos ha de entenderse lo siguiente: en los métodos y constructores los argumentos son los mismos que los argumentos de éstos; para los puntos de enlace que capturan excepciones, la excepción capturada es considerada el argumento; y para los accesos a atributos en escritura, se considera como argumento el nuevo valor a escribir sobre el atributo. Este tipo de punto de corte sirve también para capturar la información contextual en el punto de enlace (más detalles en el apartado 4.3).

Tabla 6 Pointcuts de argumentos.

Pointcut	Descripción
`args(String,..,int)`	Puntos de enlace en los que el método tenga como primer argumento un `String` y como último un `int`.
`args(SQLException)`	Puntos de enlace con un único argumento de tipo `SQLException.` Capturaría un método que tome un único argumento de tipo `SQLException`, la escritura sobre un atributo con un valor de tipo `SQLException`, o la captura de una excepción del tipo `SQLException`.

El último conjunto de *pointcuts* del lenguaje permite usar expresiones condicionales, lo cual tendrá la consecuencia de capturar sólo los puntos de enlace cuando dichas condiciones se evalúen a cierto. Toman la forma `if(`*`BooleanExpression`*`)`. La expresión lógica se evalúa en tiempo de ejecución, por los que debe ser válida en el contexto donde se sitúe. Véanse algunos ejemplos en la siguiente tabla.

Tabla 7 Pointcuts condicionales

Pointcut	Descripción
`if(Buffer.size() > MaxAllowed)`	Puntos de enlace que ocurran a partir de que el tamaño del buffer supere el valor de `MaxAllowed`.
`if(thisJoinPoint.getTarget()!=null)`	Puntos de enlace donde el objeto donde se estén ejecutando los métodos capturados no sea nulo.

Hemos examinado todas las posibilidades para definir puntos de corte en el lenguaje AspectJ. En la siguiente sección se explica el concepto de *advice*, que hará uso de los puntos de corte que hemos definido, para dotar de cierto comportamiento al programa en los puntos de enlace que se especifiquen.

4.2 AVISOS

Los puntos de corte son, junto con los avisos (*advices*), las herramientas para implementar conceptos de entrelazado dinámico en AspectJ (véase el apartado
Crosscutting dinámico, página 2). Los avisos son constructores parecidos a los métodos, donde se indica el código que se debe ejecutar en los puntos de enlace que se especifiquen en un punto de corte. Los avisos se pueden ejecutar antes, después o “alrededor” del punto de enlace. Alrededor quiere decir que puede modificarse con el aviso la ejecución del código a la altura del punto de enlace, reemplazarlo o ignorarlo, si se es eso lo que se desea. El cuerpo de un aviso se parece mucho al cuerpo de un método, ya que encapsula la lógica que debe ser ejecutada cuando se alcanza cierto punto de enlace en la ejecución. Los puntos de corte identifican *dónde* y los avisos lo completan indicando *qué* hacer.

La siguiente figura muestra una secuencia de ejecución y algunos de los posibles puntos de enlace donde se puede introducir nuevos comportamientos con avisos:

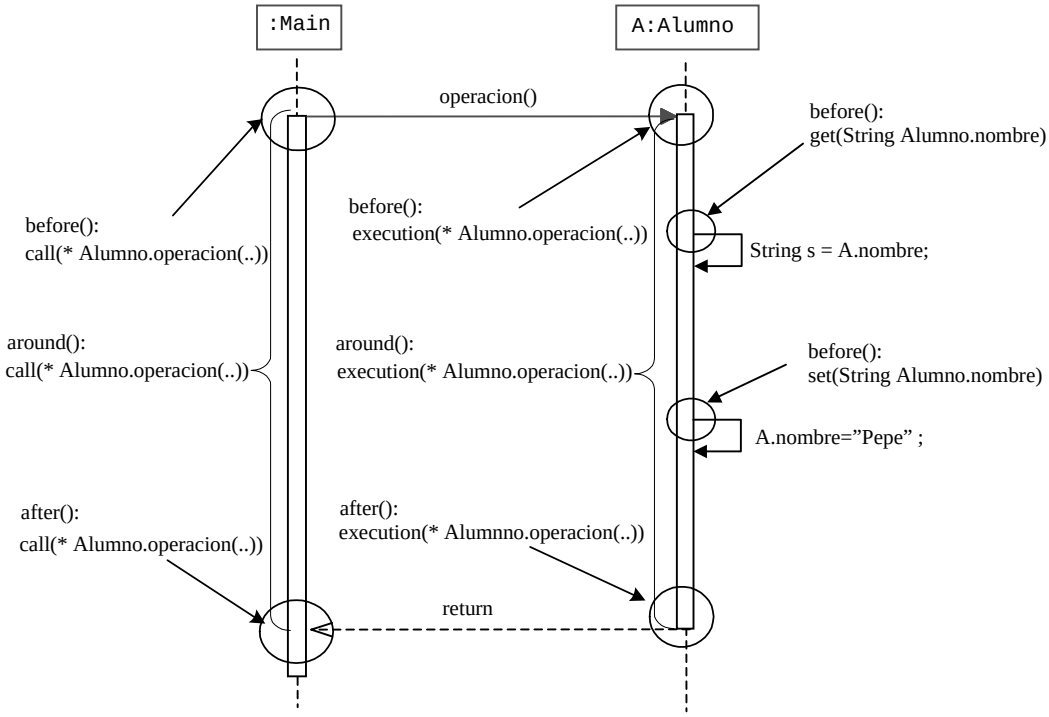


Figura 1 Diagrama de secuencia con puntos de enlace.

La figura representa un fragmento de un programa donde hay una clase `Main` y un objeto `A` de clase `Alumno`. Además se puede ver que la clase `Alumno` tiene un atributo `nombre` de tipo `String` y un método `operacion()`. En la figura se han dibujado con círculos los puntos de enlace donde los avisos `before` y `after` pueden alterar el comportamiento del programa. La parte de las líneas de vida entre círculos representa la zona de influencia de los avisos `around`. Se pueden observar los puntos de enlace de uso más habitual en AspectJ, estos son, llamadas y ejecución de funciones y acceso a atributos.

SINTAXIS DE LOS AVISOS

Los avisos pueden descomponerse en tres partes: su declaración, la definición del punto de corte (o el nombre del punto de corte que vaya a utilizar) y el cuerpo. Por claridad, los siguientes ejemplos harán uso del mismo punto de corte y que será el que a continuación se define:

```
pointcut connectionOperation(Connection connection):
  call(* Connection.*(..) throws SQLException) && target(connection);
```

El *pointcut* `connectionOperation` consta de dos puntos de corte anónimos: el primero captura todas las llamadas a métodos de la clase `Connection` que lancen la excepción `SQLException` –independientemente de los argumentos que tome o el valor que devuelva; y el segundo, `target(connection)`, captura el objeto sobre el que se hacen las llamadas. Al no ser anónimo y tener un nombre, podemos utilizarlo en la definición de los avisos cuantas veces queramos. Veamos dos ejemplos usando `before` y `around`:

```
before(Connection connection):  connectionOperation (connection){
   System.out.println("Performing operation on " + connection);
}


Object around(Connection connection) throws SQLException:
   connectionOperation(connection){
     System.out.println("Operación "+thisJoinPoint+" comenzada en "+ connection);
     proceed(connection);
     System.out.println("Operación "+thisJoinPoint+" terminada en "+ connection);
}
```

La parte anterior a los dos puntos es la declaración del aviso. En ella se especifica cuándo se ejecutará el aviso en relación con el punto de enlace, es decir, antes, después o alrededor. Además especifica la información contextual que estará disponible en el cuerpo del aviso, tal como el objeto sobre el que se ejecuta la llamada o los argumentos de esta. También habrá que indicar las excepciones que se puedan lanzar en el cuerpo del aviso, como se ha tenido que hacer en el segundo ejemplo con `SQLException`. A continuación, después de los dos puntos viene la definición del punto de corte. Pueden definirse los puntos de enlace o hacerse referencia a un punto de corte anteriormente definido, como se ha hecho en estos dos casos. Por último, el cuerpo del aviso contiene, al igual que el cuerpo de un método, las acciones a realizar y igualmente está encerrado entre llaves `{}`. La llamada al método `proceed()` en los avisos `around` ejecuta el método original capturado.

TIPOS DE AVISOS

Como se ha comentado, existen tres posibilidades para definir el momento de actuación de los avisos (`before`, `after` y `around`). Cada una de estas se explica a continuación.

Los avisos **before** se ejecutan antes de la ejecución del punto de enlace capturado. Al terminar, dan paso al método original. En el caso de que se lanzara una excepción en el cuerpo del aviso, el método original no se ejecutará. Este tipo de aviso se usa típicamente para llevar a cabo tareas de control o precondiciones, como el control de los parámetros, logging, autentificación... Véase el siguiente ejemplo:

```
before(Connection connection):  connectionOperation (connection){
     checkConnectionValidity(connection);
}
```

En este caso, el cuerpo del aviso podría comprobar mediante el método `checkConnectioValidity` y antes de llamar a las operaciones que captura `connectionOperation`, que el objeto `connection` que se va a utilizar no está corrupto.

Los avisos **after** se ejecutan después de la ejecución del punto de enlace capturado. En este caso hay que distinguir además el caso en el que un método termine con el lanzamiento de una excepción. Para ello AspectJ ofrece tres variantes para el aviso `after`: después de terminar normalmente, después de terminar lanzando una excepción y después de terminar, sea como sea (es decir, incluyendo los dos casos anteriores). La sintaxis para cada uno de esos casos es la siguiente:

```
after () : connectionOperation (connection) { ... }
```

Este aviso se ejecuta después de las llamadas capturadas por el punto de corte `connectionOperation`, ya terminen éstas normalmente o con el lanzamiento de una excepción.

```
after () returning (Object x) : connectionOperation(connection) { ... }
```

Este aviso se ejecutará después de las llamadas a los métodos capturados por el punto de corte `connectionOperation`, en el caso de que la ejecución de éstas se complete sin lanzar ninguna excepción. El valor devuelto recibe el nombre `x`, por lo que se podrá referenciar y usar en el cuerpo del aviso. Si el método termina con una excepción, el cuerpo del aviso no se ejecutará.

```
after () throwing (SQLException exc) : connectionOperation(connection){ ... }
```

Este aviso se ejecuta después de las llamadas a los métodos capturados por el punto de corte `connectionOperation`, en el caso de que éstas terminen de forma anómala devolviendo la excepción `SQLException`, según se ha indicado. A la excepción se le da el nombre `exc` en el cuerpo del aviso, por lo que así podrá utilizarse. Si el método termina normalmente, el cuerpo del aviso no se ejecutará.

Los avisos **around** envuelven el punto de enlace capturado, de tal forma que se puede anular la ejecución del punto de enlace o alterar el contexto antes de proceder. Permite también ejecutar varias veces la lógica capturada

en el punto de enlace, cada vez, por ejemplo, con diferentes argumentos. Es sin duda la más flexible de las posibilidades que ofrece AspectJ. Para permitir la ejecución por el punto de enlace capturado dentro del cuerpo del aviso, hay que utilizar una llamada a un método clave de nombre `proceed()`. Este debe ser invocado con el mismo número y tipo de argumentos que el original y devolver el mismo tipo de argumento que este. Véase el siguiente ejemplo:

```
Object around() memoryOperations(){
     try{
        proceed();
     }catch(Exception e){ /*Manejar la excepción e */ }
}
```

La intención del ejemplo es tratar de manera uniforme las excepciones que puedan lanzar las operaciones capturadas por el punto de corte `memoryOperations`. Lo que se hace es envolver la ejecución de `proceed()` entre `try/catch`, lo que permite eliminar este código de cada uno de los métodos y manejarlo aquí de manera igual para todos.

4.3 PASO DE INFORMACIÓN CONTEXTUAL

Una de las mayores utilidades en los avisos es el paso de la información contextual del punto de enlace. De esta forma se tiene acceso a información respecto a los métodos que se capturan, así como a los argumentos que estos reciben. Esto permite poder actuar de diferentes formas según el método o el tipo de los objetos capturados, así como alterar o hacer comprobaciones sobre los parámetros antes de pasar el control a los puntos de enlace. Dicha información es lo que se conoce como contexto. AspectJ dispone de una serie de herramientas para exponer dicho contexto en los avisos, como son `target()`, `this()` y `args()`. La sintaxis es muy similar a la utilizada en Java para el paso de parámetros en funciones: en la declaración de los avisos y los puntos de corte hay que especificar los parámetros con sus tipos. Dependiendo de si se utilizan puntos de corte anónimos o con nombre, la sintaxis varía ligeramente. Véanse los siguientes ejemplos, donde se ven las diferencias en los casos citados:

```
before ( Statement stmt, String query ) :
   call (* Statement.execute*( String )) && target ( stmt ) && args ( query ) {
      checkSQLQueryValidity(query);
   }
```

Este primer ejemplo muestra un aviso que hace uso de un punto de corte anónimo. En la definición de `before` se han de especificar todos los argumentos asociados con la ejecución del método capturado. En él, `target` recoge el objeto en el que se invoca el método capturado cuyo nombre ha de comenzar por `execute`, mientras que `args` captura los argumentos de dicho método. La parte del `advice` antes de los dos puntos especifica el tipo y nombre de cada uno de los argumentos que se capturan y que luego en el cuerpo pueden usarse referenciándolos con dichos nombres.

Cuando se usan puntos de corte con nombres, dichos puntos de corte deben recoger la información contextual y pasársela al aviso. El siguiente ejemplo es similar al anterior salvo que usa puntos de corte con nombre.

```
pointcut sqlQueryChecker( Statement stmt, String query ) :
   call (* Statement.execute*( String )) && target ( stmt ) && args ( query );


before (Statement stmt, String query ) :
   sqlQueryChecker (stmt, query ) {
        checkSQLQueryValidity(query);
   }
```

Funcionalmente el código es idéntico al anterior. El punto de corte `sqlQueryChecker`, además de definir los puntos de enlace, captura el contexto para que pueda ser usado en el aviso. Al igual que antes, se recoge el objeto sobre el que se invoca el método y sus argumentos. El mismo punto de corte declara el tipo y nombre de cada elemento que se recoge, tal como se declara en un método normal. La primera parte del aviso es igual que

en el caso anterior, no varía. Y a continuación se usa el punto de corte que se ha definido más arriba, haciendo coincidir los nombre de los argumentos, pero sin necesidad de indicar los tipos, pues ya se ha hecho. Otro ejemplo de paso de información contextual es el que se muestra a continuación:

```
after() returning(Connection conn ) :
   call(Connection DriverManager.getConnection(..)) {
      System.out.println("Obtenida la conexión: "+ conn );
   }
```

En él, el aviso `after returning` captura el valor devuelto por el método `getConnection` para poder ser utilizado en el cuerpo del aviso. Para ello se ha especificado el tipo y el nombre del valor devuelto en la parte `returning()` del aviso. De esta forma puede usarse el valor devuelto tal como cualquier otro objeto del contexto. Igualmente puede hacerse con el aviso `after throwing` para capturar la excepción lanzada, tal como se muestra en el siguiente ejemplo:

```
after() throwing (RemoteException ex ):
  call(* *.*(..) throws RemoteException) {
     System.out.println("Lanzada " + ex+ " al ejecutar "+ thisJointPoint );
  }
```

Haciendo uso de la API de AspectJ puede también accederse a la información contextual. Así se tienen las funciones `getThis()`, `getTarget()` y `getArgs()` que acceden a la misma información explicada antes. Sin embargo existe una diferencia: usar parámetros en la definición de los puntos de corte, como hemos visto primero, proporciona comprobación de tipos en tiempo de compilación, así como la seguridad de que tales parámetros existen. Por ejemplo, al usar el punto de corte `target(TipoX)` aseguramos que sólo se han llamado a funciones sobre un objeto de tipo `TipoX` y no de otro. Mientras que si usamos dentro del aviso la función `JoinPoint.getTarget()`, no podremos asegurar nada sobre el tipo del objeto que vamos a obtener y, ni siquiera, de si existe o no.

Otro aspecto a tener en cuenta es la devolución de valores desde los avisos `around`. Frecuentemente el valor que se devuelve se declara del mismo tipo del valor devuelto en los puntos de enlaces a los que afecta el aviso. La llamada a `proceed()` devuelve el valor retornado por el método en el punto de enlace. A no ser que se necesite manipular el valor devuelto, el aviso `around` simplemente devuelve el valor que devuelva `proceed()` –si no se invocara este método, habría que devolver un valor apropiado para la lógica del aviso. Se puede dar el caso de que el conjunto de puntos de enlaces al que afecta un `advice` tengan diferentes valores de retorno. En tal caso la solución es definir el valor devuelto por el `advice` como `Object`.

4.4 REGLAS DE ENTRELAZADO ESTÁTICO

En POA a menudo se necesita no sólo alterar el comportamiento dinámico de la aplicación, sino también modificar la estructura estática. Es lo que se conoce como entrelazado estático, frente a lo que hemos visto hasta ahora que se correspondía con el entrelazado dinámico. Hay cuatro maneras en las que AspectJ implementa lo que la terminología inglesa define como *static crosscutting*. Estas son: introducciones, modificaciones de la estructura jerárquica, errores y avisos en tiempo de compilación y suavizado de excepciones. Veamos a continuación cada una de ellas.

INTRODUCCIONES

Los aspectos a menudo necesitan introducir atributos y métodos en las clases a las que afectan. AspectJ dispone de un mecanismo llamado *introduction* para introducir dichos miembros en las clases e interfaces que se especifiquen, de forma que no sea necesario modificar las clases afectadas. Tales miembros pasan a ser parte de las clases entrelazadas y pueden tratarse como tal, con las condiciones de visibilidad que se especifiquen. La sintaxis para las introducciones es como se muestra a continuación:

```
Modifiers Type TypePattern.Id(Formals) { Body }
abstract Modifiers Type TypePattern.Id(Formals);
```

El efecto que tiene tal introducción es hacer que todos los tipos indicados en `TypePattern` dispongan de un nuevo método de nombre `Id` y cuerpo el especificado entre llaves. Igualmente se pueden introducir constructores de la siguiente forma:

```
Modifiers TypePattern.new(Formals) { Body }
```

A diferencia del ejemplo anterior, en este caso no está permitido introducir constructores en interfaces. Así que si `TypePattern` incluye una interfaz, se producirá un error de compilación. De forma similar, se pueden introducir atributos:

```
Modifiers Type TypePattern.Id = Expression;
Modifiers Type TypePattern.Id;
```

En una declaración se pueden introducir varios elementos en diferentes clases a la vez. En el siguiente ejemplo, se introducen tres métodos en tres clases diferentes, `Point`, `Line` y `Square`.

```
public String (Point || Line || Square).getName() { return name; }
```

Los tres métodos tienen el mismo nombre (`getName`), no tienen parámetros y devuelven un `String`. Los tres tendrán además el mismo cuerpo, lo cual es la gran ventaja de esta técnica, pues facilita la adaptación del código.

En cuanto a la visibilidad, AspectJ permite introducciones de miembros públicos, privados o con visibilidad de paquete (esto es, *protected*). El uso del modificador `private` implica una visibilidad privada con relación al aspecto, no necesariamente con el destino de la introducción. Así si un aspecto hace una introducción privada de un atributo en una clase, como podría ser, `private int A.x`, entonces el código en el aspecto puede hacer referencia al atributo `x`, pero ninguna otro clase puede hacerlo. Similarmente, si un aspecto hace una introducción con el modificador `protected`, tal como, `protected int A.x`, entonces cualquiera desde el paquete del aspecto (que no tiene porqué ser el mismo que el de `A`) podrá acceder al atributo `x`.

Por último comentar que si la palabra reservada `this` aparece en el cuerpo de los nuevos constructores o métodos introducidos, esta hará referencia al destino de `TypePattern` y no al aspecto donde se declara.

MODIFICACIÓN DE LA ESTRUCTURA ESTÁTICA

Muchas veces la implementación de los problemas que afectan de manera horizontal a las aplicaciones requieren actuar sobre un conjunto de clases o interfaces que comparten una base común, de tal forma que los avisos y aspectos trabajen únicamente con la API que ofrece tal tipo base. De tal forma, los avisos y aspectos serán dependientes del tipo base, en lugar de las clases e interfaces específicas de cada aplicación. Por ejemplo, un aspecto de manejo de la caché puede declarar que ciertas clases implementen la interfaz `Cacheable`. Así el aviso en el aspecto podrá trabajar solamente con la interfaz de `Cacheable`. El resultado de tal práctica es el desacoplamiento del aspecto de las clases específicas de la aplicación, haciendo así que los aspectos sean más reusables.

Con AspectJ, se puede modificar la estructura jerárquica de las clases haciendo que extiendan a una nueva clase o que implementen una lista de interfaces (siempre y cuando no se violen las reglas de herencia de Java). La sintaxis para tales declaraciones es la siguiente:

```
declare parents: [ChildTypePattern] implements [InterfaceList];
declare parents: [ChildTypePattern] extends [Class or InterfaceList];
```

Por ejemplo, el siguiente aspecto declara que todas las clases e interfaces del paquete `B` del paquete `A` tienen que implementar la interfaz `C`:

```
aspect MyAspect {
    declare parents : A..B.* implements C;
    //...
}
```

La declaración de superclases debe seguir las reglas jerárquicas que impone Java. Por ejemplo, no se puede declarar que una clase sea el padre de una interfaz. Y de igual forma, no se pueden declarar padres, de tal forma que resulte en herencia múltiple. Todo ello resultaría en errores en tiempo de compilación.

ERRORS Y WARNINGS EN TIEMPO DE COMPILACIÓN

AspectJ dispone de un mecanismo para declarar errores y advertencias en tiempo de compilación basado en los puntos de corte explicados anteriormente. De esta forma pueden implementarse comportamientos similares a los que se obtienen con las directivas de preprocesamiento `#error` y `#warning` que soportan algunos preprocesadores C y C++.

El constructor declare error permite declarar errores de compilación cuando el compilador detecta la presencia de cierto punto de enlace que se haya especificado con un punto de corte. En tal caso, el compilador informa del error mediante el mensaje proporcionado por el programador y aborta el proceso de compilación. La sintaxis del constructor es como se muestra a continuación:

```
declare error : <pointcut> : <message>;
```

De igual forma se pueden declarar advertencias mediante el siguiente constructor:

```
declare warning : <pointcut> : <message>;
```

Al afectar estas declaraciones al comportamiento en tiempo de compilación es evidente que no se pueden usar puntos de corte basados en el contexto como `this()`, `target()`, `args()`, `if()`, `cflow()`, y `cflowbelow()`.

Un uso típico de estos constructores es para definir reglas que eviten el uso de ciertos métodos no soportados o bien para limitar la localización de determinados usos del código. Un ejemplo sacado de la aplicación que sirve como ejemplo para este proyecto es el siguiente. A efectos de evitar la dispersión del código de base de datos, si limita el uso de éste al paquete `bd` y se permite, aunque avisando con un `warning`, en el paquete `aspects`.

```
pointcut dbCode(): call(Connection DriverManager.getConnection(..));
pointcut notRecommendedDbCode(): dbCode() && !within(db.*);
pointcut forbidedDbCode(): notRecommendedDbCode() && !within(aspects.*);

declare warning: notRecommendedDbCode(): "Codigo no recomendado!";
declare error: forbidedDbCode(): "Código no permitido!";
```

SUAVIZADO DE EXCEPCIONES

Es un dato estadístico el hecho de que en las aplicaciones Java aproximadamente el treinta por cierto del código corresponde al tratamiento de errores. El uso en las mismas de los mecanismos de lanzamiento de excepciones y su captura es una solución al problema, pero sin duda uno de los síntomas más evidentes de enredamiento del código. En AspectJ un aspecto puede especificar que cuando en los puntos de enlace indicados se lance cierta excepción, ésta debe saltarse el sistema usual de tratamiento de excepciones de Java y lanzarse como del tipo `org.aspectj.lang.SoftException`, que es subtipo de `RuntimeException` y que por lo tanto no necesita ser declarada. Se dice que la excepción ha sido suavizada, del inglés *softened*. La sintaxis para especificar tal comportamiento en AspectJ es la siguiente:

```
declare soft: TypePattern: Pointcut;
```

En el siguiente ejemplo, un aspecto `MyAspect` declara `Exception` en una cláusula `declare soft`, lo cual implica que cualquier excepción que se lance será encapsulada en una excepción del tipo `org.aspectj.lang.SoftException` y relanzada como tal.

```
aspect MyAspect {
  declare soft: Exception: execution(void main(String[] args));
}
```

El efecto conseguido es el mismo que se conseguiría con el siguiente aviso:

```
aspect MyAspect {
   void around() execution(void main(String[] args))
   {
      try {
        proceed();
      }catch (Exception e) {
        throw new org.aspectj.lang.SoftException(e);
      }
   }
}
```

4.5 EXTENSIÓN DE LOS ASPECTOS

Al igual que las clases, los aspectos se pueden extender, de forma que permitan la abstracción y composición de los problemas que implementan. Sin embargo, introducen algunas reglas a la hora de extenderse, tal como son:

- Un aspecto, abstracto o concreto, puede extender una clase e implementar un conjunto de interfaces.
- Las clases no pueden extender aspectos. Intentar que una clase extienda o implemente un aspecto resultará en un error de compilación.
- Aspectos que extienden aspectos. Los aspectos pueden extender otros aspectos, con lo cual no sólo se heredan los atributos y métodos, sino también los puntos de corte. Sin embargo, los aspectos sólo pueden extender aspectos abstractos. Intentar que un aspecto extienda un aspecto concreto resultará en un error de compilación.

Por ejemplo, algo usual es implementar los sistemas de log como aspectos. Así si queremos mantener un registro o historial de ciertas acciones, se puede definir un aspecto abstracto con un punto de corte abstracto, donde se indique cómo se quiere escribir la información, pero no cuales van a ser los puntos de enlace. Los aspectos que extiendan dicho aspecto, son los que concretan los puntos de corte, lo que da la posibilidad de definir clara y separadamente los elementos que se quieren controlar. El siguiente fragmento de código muestra el aspecto abstracto:

```
abstract public aspect Logger {
  abstract pointcut logPoint();

  before(): logPoint() && !within(Logger+){
    System.log(thisJoinPoint.getTarget()+"/"+thisJoinPoint.getThis());
  }
}
```

Del punto de corte se excluyen el propio aspecto y sus subclases para evitar la recursividad. El aviso recoge el objeto sobre el que se llaman las funciones y el objeto actual y los escribe en el log del sistema. El siguiente fragmento muestra un aspecto concreto que extiende a este aspecto:

```
public aspect LogDataBaseExecutions extends Logger{
  pointcut logPoint(): call(* DataBase.*.*(..));
}
```

Este aspecto proporciona una definición del punto de corte `logPoint`, que en este caso son las todas las llamadas a funciones de un supuesto paquete de base de datos `DataBase`.

4.6 INSTANCIAR ASPECTOS

A diferencia de las clases, los aspectos no se pueden instanciar con expresiones `new`. En lugar de eso, las instancias de los aspectos se crean automáticamente. Debido a que los `advices` sólo se ejecutan en el contexto de una instancia de un aspecto, el cómo se instancian los aspectos controla indirectamente cuando se ejecutan

los avisos. El criterio utilizado para determinar cómo se instancia un aspecto se hereda del aspecto al que extiende. Si éste no existiera, entonces por defecto, los aspectos siguen una política *singleton* (es decir, sólo existe una instancia de ellos). Los siguientes tres subapartados indican las diferentes posibilidades a la hora de instanciar aspectos y la sintaxis para cada una de ellas. Para ilustrarlo en cada apartado se ejecuta un ejemplo con las siguientes clases bases:

```
public class A {
  public static void main (String args[]){
      System.out.println("This     Target     Aspect");
      for (int i=0; i<3; i++)
         new B().test();
  }
}
```

La clase `A` contiene el método `main` y su cometido es crear tres instancias de la clase `B` y llamar al método `test` de éste. Primero crea una cabecera para los aspectos que se verán después. A continuación se muestra la clase `B`:

```
public class B {
  public static B b = new B();

  public void test(){
     b.metodo();
     new B().metodo();
  }

  public void metodo(){}

}
```

La clase `B` tampoco hace nada útil. Tiene un atributo estático `b` de la misma clase y dos métodos, `test` y `metodo`. El segundo de ellos será el que se capture en el punto de corte que definiremos a continuación, del que como se puede ver se producirán seis llamadas, tres de ellas con el objeto estático de la clase `B` y las otras tres con nuevos objetos creados en cada llamada a `test()`.

ASPECTOS SINGLETON

```
aspect Id { ... }
aspect Id issingleton { ... }
```

Por defecto, o usando el modificador `issingleton`, los aspectos tienen exactamente una instancia que se ejecuta para todo el programa. Dicha instancia está disponible en cualquier momento durante la ejecución del programa usando el método estático `aspectOf()` definido en los aspectos. Esta instancia se crea tal como se carga el fichero que contiene el *class* del aspecto y vive durante toda la ejecución del programa, por lo que sus avisos podrán ejecutarse en cualquier punto de enlace. Véase el siguiente ejemplo:

```
public aspect Singleton issingleton{

  pointcut m(): call (* *.metodo());

  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(Singleton.aspectOf());
  }
}
```

En este aspecto se capturan las llamadas a `metodo` y antes de ellas se imprime información del objeto actual, del objeto sobre el que se realiza la llamada y de la instancia del aspecto. Al ejecutarlo con las clases anteriores se obtiene la siguiente salida:

```
This       Target     Aspect
B@cf2c80   B@729854   Singleton@6eb38a
B@cf2c80   B@cd2e5f   Singleton@6eb38a
B@9f953d   B@729854   Singleton@6eb38a
B@9f953d   B@fee6fc   Singleton@6eb38a
B@eed786   B@729854   Singleton@6eb38a
B@eed786   B@87aeca   Singleton@6eb38a
```

Cada código diferente representa un objeto diferente. Como se ve en la primera columna, hay tres objetos, que son los tres creados por la clase `A` (de códigos `B@cf2c80`, `B@9f953d`, `B@eed786`). El método se llamó sobre cuatro objetos `B` diferentes, de los cuales el que se repite (de código `B@729854`) es el estático. Los otros corresponden a cada una de las nuevas instancias que se crean en el método `test` de la clase `B`. Y dado que se ha usado *singleton*, en todas las llamadas que capturó el punto de corte `m()` se ha usado la misma y, la única, instancia del aspecto que existía (de código `Singleton@6eb38a`). El mismo ejemplo se usará en los siguientes apartados para poder ver las diferencias.

ASPECTOS PER-OBJECT

```
aspect Id perthis(Pointcut) { ... }
aspect Id pertarget(Pointcut) { ... }
```

Si un aspecto A se define `perthis(Pointcut)` entonces existirá una instancia diferente del aspecto A cada vez que el objeto que se está ejecutando en los puntos de enlace definidos en `Pointcut` (esto es, el referenciado por `this`) sea diferente. Así, los avisos definidos en A pueden ejecutarse en cualquier punto de enlace donde el objeto actual en ejecución haya sido asociado con una instancia de A. Si no se ha creado aún una instancia de A para el objeto en cuestión, se crea una nueva. Tomando el ejemplo anterior, cambiando únicamente la política de instanciación del aspecto, tenemos el siguiente código:

```
public aspect PerThis perthis(m()){

  pointcut m(): call (* *.metodo());

  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(PerThis.aspectOf(thisJoinPoint.getThis()));
  }
}
```

En comparación con el código anterior, hay que observar la manera de obtener la instancia del aspecto. Al no haber una única instancia, no puede usarse como antes el método `aspectOf`, sino otro que tome como parámetro el objeto actual y devuelva, si es que existe, el aspecto que tiene asociado. Si no existiera tal objeto asociado se lanzaría la excepción `org.aspectj.lang.NoAspectBoundException`.

Al ejecutar el ejemplo se obtiene la salida que se muestra a continuación. En ella se ve como se tiene un aspecto diferente para cada objeto actual que está en ejecución. Hay tres instancias del objeto `test` (de códigos `B@6eb38a`, `B@eed786` y `B@2dacd1`) y para cada una de ellas se utiliza un aspecto diferente (de códigos `PerThis@9f953d` , `PerThis@87aeca` y `PerThis@ad086a` respectivamente).

```
This       Target     Aspect
B@6eb38a   B@cd2e5f   PerThis@9f953d
B@6eb38a   B@fee6fc   PerThis@9f953d
B@eed786   B@cd2e5f   PerThis@87aeca
B@eed786   B@e48e1b   PerThis@87aeca
B@2dacd1   B@cd2e5f   PerThis@ad086a
B@2dacd1   B@385c1    PerThis@ad086a
```

De manera similar, si un aspecto A es definido `pertarget(Pointcut)` entonces existirá un objeto de tipo A por cada objeto destino diferente (estos son, los referenciados por `target`) sobre el que se ejecuten los métodos capturados en los puntos de enlace definidos en el `Pointcut`. Así, los avisos definidos en A pueden ejecutarse en cualquier punto de enlace donde el objeto destino haya sido asociado con una instancia de A. O bien, si un objeto destino no tiene aún un aspecto asociado, se crea una nueva instancia de A, que será la que en adelante utilizará. Volviendo al ejemplo, tenemos esta vez el siguiente código:

```
public aspect PerTarget pertarget(m()){
  pointcut m(): call (* *.metodo());
  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
```

```
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(PerTarget.aspectOf(thisJoinPoint.getTarget()));
  }
}
```

Al igual que en el caso anterior, para obtener la instancia del aspecto es preciso indicar el objeto asociado, que, al ser *pertarget*, habrá de ser el objeto destino que se obtiene con `getTarget()`. En la salida que se obtiene al ejecutar el ejemplo se comprueba que se han creado tantos aspectos como objetos destino diferentes. A los objetos de códigos `B@cd2e5f` (este es el estático), `B@fee6fc`, `B@e48e1b` y `B@385c1` le corresponde los aspectos `PerTarget@9f953d`, `PerTarget@eed786`, `PerTarget@2dacd1` y `PerTarget@42719c`, respectivamente.

```
This       Target     Aspect
B@6eb38a   B@cd2e5f   PerTarget@9f953d
B@6eb38a   B@fee6fc   PerTarget@eed786
B@87aeca   B@cd2e5f   PerTarget@9f953d
B@87aeca   B@e48e1b   PerTarget@2dacd1
B@ad086a   B@cd2e5f   PerTarget@9f953d
B@ad086a   B@385c1    PerTarget@42719c
```

Al usar alguna de las dos técnicas anteriores (`perthis` o `pertarget`) hay que tener en cuenta la siguiente restricción a la hora de escribir el código: en los aspectos así definidos, en un punto de enlace donde el código fuente no esté disponible para el objeto actual, no se instanciará el aspecto definido `perthis` o `pertarget` para dicho punto de enlace. De tal modo que, los aspectos definidos `perthis(Object)/ pertarget(Object)` no crearán instancias del aspecto para cada objeto, sino sólo para aquellos para los que el compilador controla su código *class*.

ASPECTOS PER-CONTROL-FLOW

```
aspect Id percflow(Pointcut) { ... }
aspect Id percflowbelow(Pointcut) { ... }
```

Si un aspecto A se define `percflow(Pointcut)` o `percflowbelow(Pointcut)` entonces un objeto de tipo A se crea cada vez que el control pasa a uno de los puntos de enlace capturados por el `Pointcut`, bien tal como el flujo de control pasa al punto de enlace o a partir de éste, pero excluyéndolo, respectivamente. Véase el anterior ejemplo con una política de `PerCflow` y los resultados:

```
public aspect PerCflow percflow(m()){
  pointcut m(): call (* *.metodo());
  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(PerCflow.aspectOf());
  }
}
```

En este caso el método `aspectOf()` no necesita ningún parámetro. AspectJ mantiene una pila de aspectos *percflow*, que se crean al entrar al llegar el flujo de control al punto de enlace y se destruyen al salir de él, por lo que se asegura que siempre se mantienen el número de instancias correctas. Durante la ejecución, los aspectos se toman de esta pila.

En el resultado que se muestra a continuación puede verse como se ha creado una instancia diferente del aspecto cada vez que se ha realizado una llamada y puesto que se realizaron seis, se tienen seis en total, indiferentes del objeto actual o del destino.

```
This       Target     Aspect
B@cd2e5f   B@9f953d   PerCflow@fee6fc
B@cd2e5f   B@eed786   PerCflow@87aeca
B@e48e1b   B@9f953d   PerCflow@2dacd1
B@e48e1b   B@ad086a   PerCflow@385c1
B@42719c   B@9f953d   PerCflow@30c221
B@42719c   B@19298d   PerCflow@f72617
```

Como en los ejemplos anteriores hay tres instancias del objeto actual `B` y cuatro como `target` (al contar con el objeto estático). Sin embargo, esta vez, las instancias del aspecto son diferentes para cada una de las llamadas.

ASPECT PRIVILEGE

```
privileged aspect Id { ... }
```

El código escrito en los aspectos está sujeto a las mismas reglas de control de acceso que el código Java en lo que respecta a los miembros de las clases o los aspectos. Así, por ejemplo, el código escrito en un aspecto no puede hacer referencia a miembros con visibilidad *protected*, a no ser que el aspecto esté definido en el mismo paquete. Mientras que estas restricciones son aceptables para muchos aspectos, puede haber casos en los que los aspectos necesiten en sus *advices* o *introductions* acceder a recursos que son privados o protegidos. Para permitir esto, los aspectos tienen que ser declarados `privileged`. En el código de los miembros `privileged` se tiene acceso a todos los miembros, incluidos los privados. En el siguiente ejemplo se muestra un ejemplo de un aspecto que gracias a ser declarado como *privileged* puede acceder a un atributo privado de la clase que utiliza. La clase `Divisor` contiene un atributo privado y un método que realiza la división dado un entero como parámetro. El aspecto chequea que el divisor no sea cero.

```
class Divisor {
   private int i;
   int divide(int x) { return x/i; }
}


Divisor(int value){i = value;}

privileged aspect C {
   before(Divisor d): call(int Divisor.divide(int)) && target(d) {
      if (d.i == 0) throw new RuntimeException();
   }
}
```

Si C no hubiese sido declarado `privileged`, el intento de referenciar a `d.i` hubiera implicado un error de compilación como el siguiente:

```
C.java:4:13: Divisor.i has private access
     if (d.i == 0) throw new RuntimeException();
```

4.7 ASPECTOS DOMINANTES

```
aspect Id dominates TypePattern { ... }
```

Un aspecto puede declarar que los *advices* que define dominan sobre los *advices* en otros aspectos, lo que implica que el aviso dominante tiene mayor precedencia que los especificados en los aspectos que coincidan con `TypePattern`. Por defecto en AspectJ no existe ningún orden definido, por lo que, si se precisa ejecutar los avisos en determinado orden, es necesario especificarlo con la cláusula `dominates`. La semántica es que si un aspecto `A` domina sobre un aspecto `B`, entonces los avisos de `A` tienen más prioridad y se ejecutan antes que los de `B`. En cuanto a las situaciones de conflicto como puede ser la siguiente, el compilador 1.0.6 no informa del error y simplemente actúa ignorando una de las dos.

```
aspect A dominates B{…}
aspect B dominates A{…}
```

Hay que advertir que en la versión 1.1 de AspectJ la sintaxis de `dominates` ha cambiado, desapareciendo `dominates` y pasando a usarse la siguiente declaración:

```
declare precedence ":"  TypePatternList ";"
```

Por poner un ejemplo, si quisiéramos definir una regla de precedencia, de modo que los avisos de los aspectos cuyo nombre contenga la cadena `Security` precedan a cualquier otro y que además, los avisos del aspecto `Logging` y sus subclases precedan a los restantes, entonces deberíamos escribir lo siguiente:

```
declare precedence: *..*Security*, Logging+, *;
```

Sería un error en una sentencia `precedence` que un aspecto pudiera ser referido por más de un patrón de tipos. Así por ejemplo, la siguiente declaración produciría un error:

```
declare precedence:  A, B, A ;
```

Sin embargo no es un error tener en múltiples sentencias este tipo de problema de circularidad, como puede producirse con las siguientes dos declaraciones:

```
declare precedence: B, A;
declare precedence: A, B;
```

Y de hecho un sistema así definido puede ser correcto, siempre y cuando los avisos de `A` y `B` no compartan puntos de enlace. De esa forma queda claramente expresado que `A` y `B` son independientes.

Esta técnica se ha utilizado en la aplicación que sirve de ejemplo a este proyecto en los aspectos `Pooling` y `ConnectionChecking`, donde el primero de ellos es responsable de mantener un conjunto de conexiones a la base de datos. En caso de hacerse una llamada en la aplicación para crear una nueva conexión, ésta es capturada por el aspecto, el cual le proporciona una conexión de entre las disponibles (si es que las hay). Debido a que el estado de las conexiones almacenadas puede no ser correcto, es necesario, antes de servir estas conexiones, comprobar su validez. De eso se encarga el otro aspecto, `ConnectionChecking`, que ejecuta sus avisos, necesariamente, antes que los de `Pooling`. A continuación se presenta el código de estos dos aspectos (de forma abreviada, pues el código completo puede verse en los ficheros que acompañan a este proyecto):

Código ConnectionChecking.java

```
public aspect Pooling
{

  private static Stack pool = new Stack();

  pointcut poolGet(): call(static Connection DriverManager.getConnection(..));
  pointcut poolPut(): call(void Connection.close());


  Connection around() throws SQLException: poolGet()
  {
    synchronized(pool)
    {
      if( pool.empty() )
         return proceed();
      return (Connection)pool.pop();
    }
  }


  void around(): poolPut()
  {
    Connection conn = (Connection)thisJoinPoint.getTarget();
    pool.push(conn);
  }

}
```

Código ConnectionChecking.java

```
public aspect ConnectionChecking dominates Pooling
{

  Connection around() throws SQLException:
     call(static Connection DriverManager.getConnection(..))
  {
      Connection conn;
      do{
        conn = proceed();
      }while(badConnection(conn));

      return conn;
  }

  private boolean badConnection(Connection conn)
  {...}
}
```

4.8 LA API DE ASPECTJ

AspectJ viene con dos paquetes en su API: `org.aspectj.lang` y `org.aspectj.lang.reflect`. Se puede consultar la documentación completa en la distribución, por lo que aquí solo se comentarán de forma breve las partes que se usan más frecuentemente.

Interfaces JoinPoint y JointPointStaticPart
El compilador de AspectJ crea dos variables especiales para cada punto de enlace que se encuentra: `thisJoinPoint` y `thisJoinPointStaticPart`. Estas variables tienen los tipos que implementan `org.aspectj.lang.JoinPoint` y `org.aspectj.lang.JoinPoint.StaticPart`, respectivamente. La diferencia entre ambas, es que esta última contiene información que no depende del contexto en ejecución. Así podemos tener información sobre los puntos de enlace, como la signatura de los métodos, la localización del código, el tipo de punto de enlace (method-execution, call-execution, ...) Y respecto a la información contextual en tiempo de ejecución, podemos tener el objeto actual que se esté ejecutando, el objeto sobre el que se llaman los métodos o los argumentos disponibles en el punto de enlace.

Interface Signature y sus subinterfaces
La interfaz `org.aspectj.lang.Signature` declara métodos para describir los constructores del lenguaje en los puntos de enlace (métodos, atributos, constructores, excepciones...) Cada una de las subinterfaces añade métodos relevantes a cada tipo de signatura particular. Se tienen métodos para obtener los nombres de los métodos o los atributos, los nombres y los tipos de los parámetros de los métodos capturados, los tipos de las excepciones que pueden ser lanzadas, el tipo del valor devuelto por los métodos o los advices...

Interface SourceLocation
La interfaz `org.aspectj.lang.reflect.SourceLocation` ofrece métodos para localizar físicamente la posición de los puntos de enlace en el código fuente. Dispone de métodos para obtener el nombre del fichero que contiene el punto de enlace, así como la línea y la columna en la que está.

Exceptions
La API de AspectJ define tres expcepciones, de nombres: MultipleAspectsBoundException, NoAspectBoundException y SoftException.

5 HERRAMIENTAS DEL LENGUAJE

En los siguientes apartados se presentan las herramientas que acompañan la distribución de AspectJ, a saber, el compilador del lenguaje, la herramienta de documentación y el sistema de navegación por las clases y aspectos. Tan solo se describirán los parámetros más usados, pues para una información completa puede referirse a la documentación de AspectJ.

5.1 COMPILADOR DEL LENGUAJE

El compilador para el lenguaje AspectJ se llama `ajc`. Puede lanzarse desde la línea de comandos o ser invocado desde una aplicación java, ya que el mismo compilador está escrito en Java y puede ejecutarse usando la clase `org.aspectj.tools.ajc.Main` (de este modo se tienen los mismos parámetros que de la manera anterior). El formato general de las llamadas al compilador es el siguiente:

```
ajc [option] [file...| @file... | -arglist file...]
```

La herramienta `ajc` compilará los ficheros listados en la línea de comandos o los ficheros que se encuentre en el fichero indicado. Por comodidad puede usarse un fichero con la lista de opciones y los nombres de los ficheros a compilar, el cual se especifica con el símbolo "`@`" o con la opción `–arglist`. En dicho fichero, cada línea debe contener una sola opción o el nombre de un único fichero. Todos los ficheros que se quieran compilar tienen que ser indicados, debido a que `ajc` no busca en el *class path* las clases que se necesitan. Entre las opciones más usuales que acepta `ajc` están las siguientes:

- **-argfile <file>** Indica un fichero con los argumentos que ha de tomar el compilador.
- **-classpath <path>** Indica dónde encontrar las clases de los ficheros de los usuarios.
- **-d <dir>** Indica donde guardar los .class generados.
- **-encoding <encoding>** Indica la codificación de los caracteres usada en los ficheros.
- **-preprocess** No genera los `.class` si no sólo los `.java`.
- **-workingdir <dir>** Indica donde indicar los ficheros .java preprocesados.
- **-usejavac** Indica al compilador que use `javac` para general los ficheros `.class`.
- **-verbose** Muestra mensajes por pantalla de lo que esté haciendo el compilador.
- **-version** Muestra la versión del compilador.

5.2 GENERADOR DE DOCUMENTACIÓN

La herramienta `ajdoc` genera documentación en formato HTML acerca de la API. Está basada en la herramienta `javadoc` y acepta todos los mismos parámetros estándar que ésta acepta. Pero además de documentar las clases, `ajdoc` documenta los puntos de corte, los avisos y las declaraciones; en los *advices* e *introductions* aparecen enlaces a las clases afectadas; enlaza los miembros introducidos con las clases que los introducen; y los métodos afectados por avisos con los correspondientes avisos en los aspectos. Al igual que `ajc`, `ajdoc` necesita que, bien por la línea de comandos o mediante un fichero, se listen todos los ficheros fuentes que tiene que documentar. Sin embargo, a `ajdoc`, pueden indicársele nombres de paquetes. La forma general de usar `javadoc` es como sigue:

```
ajdoc [options] [packagenames] [sourcefiles] [@files]
```

-public Documenta solo las clases y los miembros públicos.
-protected Documenta las clases y miembros protegidos y públicos.
-package Documenta las clases y miembros protegidos, públicos y con visibilidad de paquete.
-private Documenta todas las clases y miembros.
-help Muestras las opciones.
-sourcepath <pathlist> Indica donde encontrar los ficheros que contienen el código fuente.
-classpath <pathlist> Indica dónde encontrar los ficheros *class* de usuario.
-d <dir> Indica el directorio destino de los ficheros de salida.
-verbose Muestra mensajes de lo que `javadoc` está haciendo mientras se ejecuta.
-version Imprime la versión de `javadoc`.

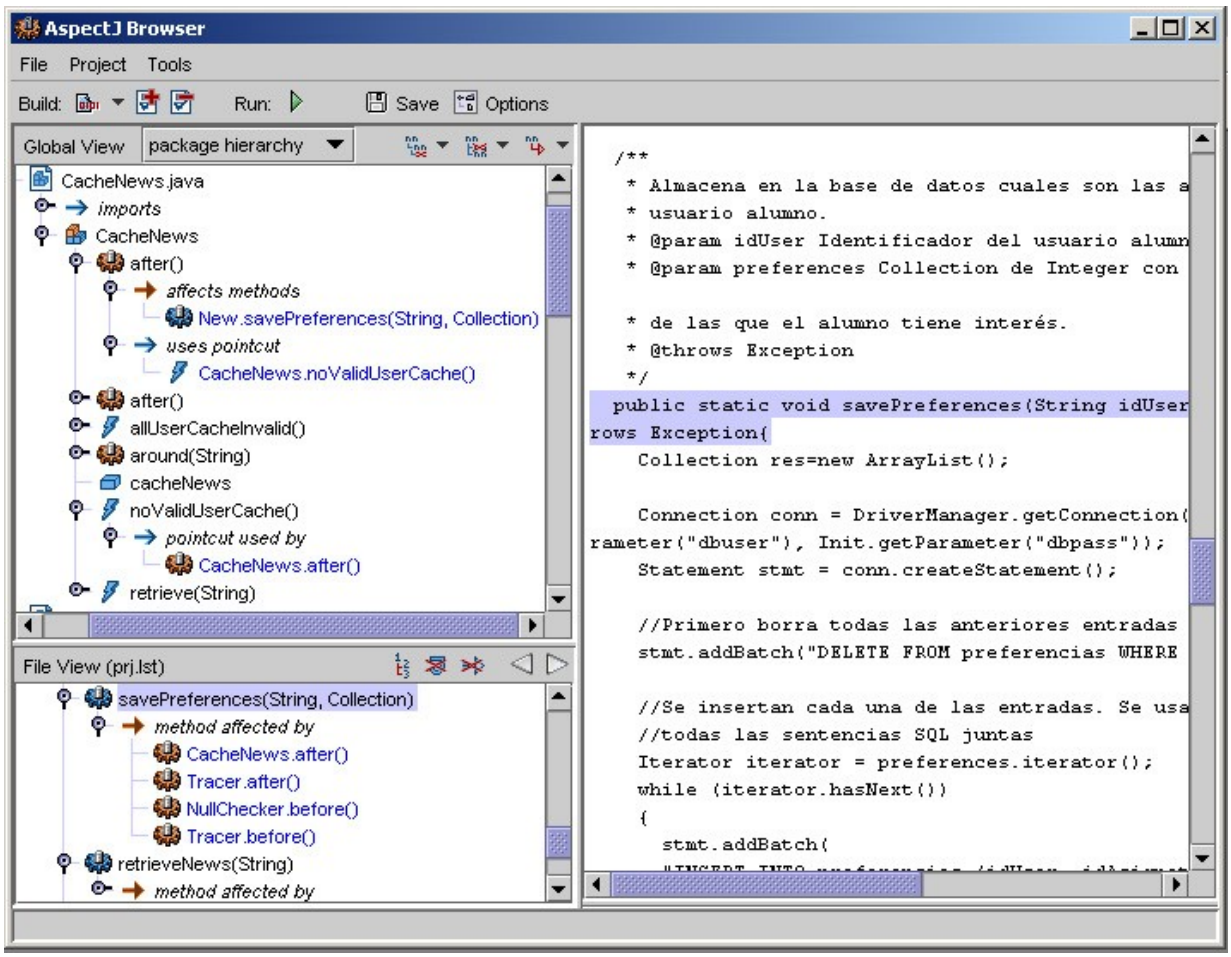
5.3 NAVEGADOR DE ASPECTJ

Otra de las herramientas que nos encontramos en el directorio `/bin` de AspectJ es `ajBrowser`. Esta ofrece un GUI (graphic user interface) para compilar programas con `ajc` y navegar por la estructura de la aplicación. Además permite editar el código de los ficheros y después de compilar, ejecutar el programa.

Para ejecutarlo, bien se escribe desde la línea de comandos `ajbrowser` (para invocar el *script* del subdirectorio `/bin`), o bien se llama usando `aspectjtools.jar`. No es preciso indicar ningún parámetro, pero pueden indicarse ficheros de configuración con las opciones y ficheros que se desean cargar (de extensión `.lst`). Haciendo uso de `aspectjtools.jar` un ejemplo sería:

```
java -jar <directorio de AspectJ>/lib/aspectjtools.jar <mi aplicación>/list.lst
```

De momento AjBrowser tiene soporte para Eclipse, JBuilder, SunONE/Forte y Emacs, a parte de poder usarse independientemente. En la Pantalla 1 se muestra una captura de `ajBrowser`, donde se han cargado los ficheros de la aplicación que sirve de ejemplo al proyecto.



Pantalla 1 AspectJ Browser

Para navegar por la estructura, se muestra a la izquierda un listado de las clases y aspectos cargados. En la parte superior, de cada aspecto se muestran los puntos de corte que define, así como los avisos que los utilizan y, éstos, a los métodos que afectan. En el cuadro superior izquierda, se ve como un aspecto de nombre `CacheNews`, define un *pointcut* `noValidUserCache()`, que es usado por un aviso `after` que afecta al método `savePreferences(String, Collection)` de la clase `New`. Abajo, puede verse desde la perspectiva contraria. Navegando por la clase `New` y el método `savePreferences`, podemos ver todos los aspectos que le afectan. Entre ellos, el primero que aparece es `CacheNews.after()`, que es el comentado antes, pero además se puede ver que hay otros tres.

Esto concluye el apartado referente a Aspect/J. El siguiente de los apartados establece una comparación entre los dos lenguajes de aspectos de propósito general que hemos visto, reflejando de manera clara sus diferencias conceptuales y sus objetivos.

6 ASPECTJ VS. HYPER/J

En los apartados anteriores se han analizado dos de los lenguajes que más éxito están teniendo en el paradigma de la POA y que, como se ha podido ver, no enfrentan el problema de la misma forma. El objetivo de la POA es identificar y separar conceptos no funcionales que afectan horizontalmente a la descomposición dominante de una aplicación.

Para conseguir este objetivo AspectJ extiende Java aumentando el modelo OO definiendo nuevos constructores que permiten la definición de aspectos capaces de encapsular en módulos separados conceptos dispersos por la estructura de clases, así como constructores para definir los puntos de enlace donde las clases o métodos han de ser modificados. Cada aspecto puede afectar a varias clases, lo que permite localizar de manera efectiva la implementación de los aspectos. El principal inconveniente hasta ahora es la dependencia de los aspectos con las clases a las que afectan, pues, por lo general, sin hacer referencia a éstas no se entienden fácilmente. La idea de Hyper/J es soportar la integración de módulos, recogidos en lo que en la terminología se denominan *hyperslices*, donde cada una de ellos encapsula una competencia específica. Los *hyperslices* pueden solaparse o existir de manera independiente unos de los otros. En lugar de definir puntos de enlace, existen reglas de composición para componer los *hyperslices*, de modo que, siguiendo una regla general de composición establecida en el lenguaje y las reglas específicas particulares para cada caso especial, combinan sus funcionalidades. Así que tenemos que, mientras que AspectJ es una extensión al lenguaje Java, Hyper/J es una herramienta que utiliza java como lenguaje.

Empezando con las diferencias, tenemos las reglas de composición. En AspectJ no hay como en Hyper/J una regla de composición general. En su lugar, cada aspecto contiene sus reglas, indicando particularmente la manera en la que dicho aspecto debe enlazarse con las clases bases. A diferencia, esto se hace en Hyper/J en ficheros separados –como hemos visto– lo cual permite de veras combinar aplicaciones que se escribieron sin tener en mente la POA. En el caso de AspectJ podemos decir que, habiendo conseguido encapsular aspectos, no encapsula las reglas de entrelazado, que son el nuevo *tangled code* (código disperso), problema que no tenemos usando Hyper/J.

La principal diferencia conceptual entre ambos parte de la existencia en AspectJ de un programa “base” en el cual los aspectos se enlazan. En Hyper/J no existe el concepto de un programa base, sino que cada componente de código es independiente y provee de forma completa el código para la unidad particular de descomposición. Esa es la principal ventaja de Hyper/J, y es el hecho de utilizar componentes verdaderamente independientes entre sí, lo cual beneficia la reusabilidad de los mismos (incluidas dos aplicaciones que se diseñaron por separado y que son capaces de funcionar independientemente).

Otra diferencia es que AspectJ trabaja con el código fuente, mientras que Hyper/J utiliza el bytecode de los ficheros class. Por tanto no necesitamos el código fuente, aunque sí conocer la interfaz (nombres de las clases, funciones...), lo cual se puede obtener fácilmente con compiladores inversos o con la documentación. Un problema de Hyper/J hoy en día es que es aún una herramienta en desarrollo, mientras que AspectJ tiene ya disponible su versión 1.1.

AspectJ es una herramienta ideal para incluir cierto comportamiento en muchos sitios diferentes de una aplicación. Sin embargo, su uso se complica cuando se trata de combinar grandes componentes con muchas clases. Habría que distinguir entre el código base y los muchos aspectos que identificáramos y entonces combinarlos. En el caso de tener muchos aspectos, si quisiéramos realizar distintas combinaciones y probarlas repetidas veces, entonces esto se convierte en un proceso muy laborioso. Y se debe a que en AspectJ cada aspecto es una entidad distinta y separada, y con muchos aspectos, se pierde cohesión. Por el contrario, en Hyper/J se tiene el concepto de hipersección. Un hyperslice contiene todas las clases y métodos asociados con una competencia específica. Por lo tanto añadir o quitar una de estas competencias implica un solo movimiento, respectivamente añadir o quitar la hipersección concreta.

De cualquier modo, el modelo propuesto por AspectJ ha tenido un mayor éxito entre la comunidad informática. Puede encontrarse mucha más información y empresas que lo están utilizando para sus proyectos. Aparecen

nuevos lenguajes cuyos modelos se basan en el de AspectJ, como es el caso de AspectC#, AspectC++ o AspectS, entre otros. Las nuevas versiones por venir y una mayor documentación pueden hacer ver un auge en Hyper/J.

7 APÉNDICE (EJEMPLO CON UN PAQUETE MATEMÁTICO)

Supongamos un pequeño paquete matemático que nos permite implementar sumatorios y potencias. Con objeto de que sea más legible, no se contemplan los casos particulares en las potencias, como ocurre cuando el cero o el uno intervienen. Cada tipo de operación está representada con una clase y cada una de ellas implementa un método `calcula` que es el encargado de realizar los cálculos.

Implementación del ejemplo

Código 1 Potencia.java

```
package math;

public class Potencia{

    int base;
    int exponente;

    public int getBase(){
        return base;
    }

    public int getExponente(){
        return exponente;
    }

    public void setBase(int dat){
        this.base=dat;
    }

    public void setExponente(int dat){
        this.exponente=dat;
    }

    public int calcula(){
        int result=0;
        for (int i=0; i<exponente; i++)
            result=result+base*base;

        return result;
    }
}
```

Código 2 Sumatorio.java

```
package math;

public class Sumatorio{

    int iteraciones;
    int sumando;

    public Sumatorio(int iteraciones, int
sumando){
        this.iteraciones = iteraciones;
        this.sumando = sumando;
    }

    private int getSumando(){
        return sumando;}

    private int getIteraciones(){
        return iteraciones;
    }

    public int calcula(){
        int result=getSumando();

        for (int i=0;
i<this.getIteraciones()-1; i++)

result=result+this.getSumando();

        return result;
    }
}
```

Para probar esta sencilla librería podemos utilizar el siguiente código de ejemplo contenido en un fichero `main.java`. En el se crean dos objetos, uno de cada una de las clases y después se llama a sus respectivos métodos `calcula()`:

Código 3 main.java

```
import math.*;

public class main{

public static void main(String args[])
{
   System.out.println("Comienzo de main");

   Potencia pot = new Potencia();
   pot.setBase(2);
   pot.setExponente(4);
```

```
   System.out.println(pot.getBase()+ " elevado a " + pot.getExponente()+" es");
   System.out.println(pot.calcula());

   Sumatorio sum = new Sumatorio(2,3);

   System.out.println("2 sumado 3 veces es"+ sum.calcula());
  }
}
```

Supongamos que estas operaciones para el cálculo del sumatorio y las potencias son altamente costosas y que decidimos implementar una política por la que los cálculos se lleven a cabo sólo si el procesador tiene el tiempo suficiente. En lugar de tener que modificar cada una de las clases, podemos crear un aspecto que implemente este nuevo requisito, de modo que no sea necesario cambiar, ni complicar el código ya escrito. Además, para el matemático que ha creado las clases anteriores, ¿qué tiene que ver con él ahora este nuevo comportamiento?. Sin duda, la solución que nos ofrece la POA es de utilidad en estos casos. Veamos cómo se haría:

Código 4 CheckTimeToProcess.java [1]

```
public aspect CheckTimeToProcess{

     pointcut processPoint(): execution(int math.*.calcula(..));

     int around(): processPoint()
     {
        while(!timeToProcess()){
           //sleep();
        }

        System.out.println("Comenzando calculos..");
        int result = proceed();
        System.out.println("Calculos terminados!");
        return result;
     }

     public boolean timeToProcess()
    {
        System.out.println("Comprobando tareas.. ");
        return true;
     }
}
```

Lo que se ha hecho es definir un punto de corte para indicar que queremos capturar los puntos donde se llaman a las funciones `calcula` del paquete `math` y un aviso, que utiliza dicho corte, para comprobar si se dan las condiciones para poder realizar los cálculos o no. Antes de ejecutar `calcula`, se chequeará la condición `timeToProcess` (que contendría la lógica para ver la ocupación del sistema) y en caso de ser afirmativa, se procede con el cálculo (eso se consigue con la llamada a `proceed`).

Se necesita establecer un acuerdo con el lenguaje base a fin de facilitar la definición de los puntos de enlace. En este caso se ha decidido que las funciones que realicen el cálculo pesado se llamen `calcula`. De este modo es fácil identificar dichos puntos en el código[2].

Además para fines de depuración queremos mostrar por pantalla todas las ejecuciones de las funciones públicas `get` y `set` de las clases del paquete `math` (para dar un poco de juego, observe que en `Sumatorio`, estos métodos no son públicos, así que no entran dentro del conjunto de puntos de enlace que queremos definir). Para ello escribiremos el siguiente aspecto en un fichero `Logger.java`:

[1] Por simplicidad, la funcionalidad no está totalmente implementada, pero se entiende qué es lo que tendría que hacerse en el cuerpo del bucle while y la operación timeToProcess.

[2] Se podría obligar a las clases a que tuvieran el método calcula, haciendo que las clases implementaran una clase abstracta Operaciones, que tuviera dicha función.

Código 5 Logger.java

```
public aspect Logger{
   pointcut logPoint(): execution(public * math.*.get*(..))
                     || execution(public * math.*.set*(..));

   before(): logPoint(){
       System.out.println(thisJoinPoint);
   }
}
```

Aquí se ha definido un corte `logPoint` que declara los puntos de enlace donde se llaman a las funciones públicas `get` y `set` de cualquiera de las clases del paquete `math`. Observe que para definir los patrones de los nombres de las funciones se puede utilizar el carácter especial asterisco `*` para especificar que el nombre puede terminar en cualquier cadena de caracteres. Además en un punto de corte se pueden encadenar la definición de varios puntos de enlace usando los símbolos `||`. La sintaxis es mucho más rica aún, pero se verá con detalle en los siguientes capítulos.

	Secuencia de llamadas	**Pointcut**	**Advice**	**Salida**
1	main(String args[])			Comienzo de main
2	void math.Potencia.setBase(int)	logPoint()	before	execution(void math.Potencia.setBase(int))
3	void math.Potencia.setExponente(int)	logPoint()	before	execution(void math.Potencia.setExponente(int))
4	int math.Potencia.getBase()	logPoint()	before	execution(int math.Potencia.getBase())
5	int math.Potencia.getExponente()	logPoint()	before	execution(int math.Potencia.getExponente())
6	timeToProcess()	processPoint()	around	Comprobando carga procesador...
7		processPoint()	around	Comenzando cálculos...
8	int math.Potencia.calcula()			
9		processPoint()	around	Calculos terminados!
10	main(String args[])			2 elevado a 4 es 16
11	.....	.....	.....	.....

Y bien esto es todo lo que se requiere escribir. Cuando se compile junto con las clases de la aplicación, el entrelazador de código se encargará de insertar estos nuevos comportamientos, sin que se haya requerido, por tanto, más intervención del programador. Si no se quisieran seguir utilizando, lo único que hay que hacer es volver a compilar la aplicación pero sin el aspecto que se quiera excluir (¡sin necesidad de retocar código!)

La variable `thisJoinPoint` la crea el compilador de AspectJ y contiene meta-información sobre el punto de enlace que se haya alcanzado. Es usual utilizar los métodos, `getThis()`, que devuelve el objeto sobre el que se estuviera ejecutando la función (si es que está disponible); `getTarget()`, que igualmente si está disponible representa al objeto capturado; `getSignature()`, que devuelve el nombre de la función a la que se refiere el punto de enlace; y `getArgs()` que devuelve los argumentos disponibles en el punto de enlace. Al ejecutarlo se obtiene el siguiente resultado por pantalla:

```
Comienzo de main
execution(void math.Potencia.setBase(int))
execution(void math.Potencia.setExponente(int))
execution(int math.Potencia.getBase())
execution(int math.Potencia.getExponente())
Comprobando carga del procesador...
Comenzando calculos...
Calculos terminados!
2 elevado a 4 es 16
Comprobando carga del procesador...
Comenzando calculos...
Calculos terminados!
2 sumado 3 veces es 6
```

Las cuatro primeras líneas muestran el momento previo a las llamadas a las funciones públicas de la clase Potencia, `setBase`, `setExponente`, `getBase` y `getExponente`. En el caso de `sumatorio`, las funciones `get` son privadas, por lo que no se muestran, ya que tales puntos de enlace no están definidos en el punto de corte `logPoint`. Nuestro otro aspecto nos ha avisado por pantalla del comienzo del proceso de cálculo, esto es de las llamadas a `calcula`, puesto que el procesador tenía tiempo libre. Para ver más claramente cual es la secuencia de llamadas podemos observar la tabla anterior donde se ha querido mostrar lo siguiente[3]:

- En la primera fila vemos el comienzo del método `main` y la salida que produce.
- Después de crearse el objeto `Potencia`, se llama a sus métodos `sets` y eso hace que se ejecute el aviso `before` ya que el punto de corte captura dichas llamadas. Eso se muestra en las filas tres y cuatro.
- Igualmente ocurre con los métodos `gets` en las filas cuatro y cinco.
- Cuando se llega al método `calcula()`, se ejecuta el aviso `around`, que hace que primero se llame a la función `timeToProcess()` (línea 6) y después se produce la salida mostrada en la línea siete que corresponde al cuerpo del aviso.
- A continuación se ejecuta el cuerpo de `calcula()`, que se muestra en la fila ocho y que no implica ninguna salida.
- Después de ejecutarse, se vuelve al aviso `around` y se produce la salida mostrada en la línea nueve.
- De vuelta en el método `main`, se produce la salida que muestra los resultados.

Proceso de entrelazado

El compilador de AspectJ nos permite ver los resultados intermedios que genera antes de producir el byte-code final. Esto nos permite entender mejor el funcionamiento del entrelazador de código y ver cómo funciona la implementación que el grupo de AspectJ ha hecho de los conceptos de la metodología de POA. Para ello basta con compilar los ficheros con la directiva `–preprocess`. Para compilar el ejemplo anterior obteniendo el código preprocesado, debemos escribir:

```
>ajc -preprocess main.java Logger.java checkTimeToProcess.java ./math/Sumatorio.java
./math/Potencia.java
```

Como resultado el compilador ajc produce en un directorio de trabajo (que también se podría haber especificado en la línea de comandos) un conjunto de ficheros Java algo más grandes y menos legibles que los originales y que incorporan los aspectos especificados entrelazados. En el caso del ejemplo, los ficheros que se han creado son los mismos, incluidos los que describen los aspectos. El siguiente listado muestra el nuevo fichero `logger.java` generado por el compilador:

Código 6 logger.java

```
public class Logger {

  public final void before0$ajc(org.aspectj.lang.JoinPoint thisJoinPoint){
    System.out.println(thisJoinPoint);
  }

  public Logger() {
    super();
  }

  public static Logger aspectInstance;

  public static Logger aspectOf() {
    return Logger.aspectInstance;
  }

  public static boolean hasAspect() {
    return Logger.aspectInstance != null;
  }
```

[3] Por brevedad no se ha mostrado la traza completa, pues no aporta ninguna idea nueva.

```
  static {
       Logger.aspectInstance = new Logger();
  }
}
```

La primera observación es que se ha convertido en una clase normal java y ha dejado de estar definido como un aspecto. Segundo, el cuerpo del aviso `before`, se ha convertido en una función normal java de nombre `before0$ajc` que recibe un parámetro del tipo `org.aspectj.lang.JoinPoint` con información contextual. El resto es código auxiliar necesario para la implementación.

El código generado para `Potencia.java` es bastante más largo que el original, ya que, además, se han añadido más métodos. Como ejemplo veremos un fragmento para ver que ha ocurrido con el método `getBase()`:

```
public int getBase() {
  final org.aspectj.lang.JoinPoint thisJoinPoint =org.aspectj.runtime.reflect.Factory.makeJP
       (Potencia.getBase$ajcjp1, this, this, new java.lang.Object[] {});

  Logger.aspectInstance.before0$ajc(thisJoinPoint);

  return this.base;
}
```

Este método ha conservado su nombre, pero ha variado su cuerpo. Se ve afectado por el aviso `before` del aspecto `Logger`, lo que se realiza en la segunda instrucción con la llamada a `Logger.aspectInstance.before0$ajc(thisJoinPoint)`. A continuación se ejecuta el cuerpo original que tuviera el método, en este caso únicamente la instrucción `return`. La primera instrucción facilita la información contextual, creando el objeto `thisJoinPoint` que recibe como primer parámetro un atributo estático que el compilador le ha añadido a la clase, `org.aspectj.lang.JoinPoint.StaticPart`, así como el propio objeto y la lista de parámetros de la llamada, en este caso, una lista de objetos vacíos.

Algo parecido ha ocurrido con el método calcula de las clases `Sumatorio` y `Potencia` que estaba afectado por un aviso `around`. El compilador ha generado principalmente tres métodos para ella en cada clase para entrelazar el código del aspecto `checkTimeToProcess`. A continuación se muestran estas funciones para la clase `Potencia`:

```
public int calcula() {
  return this.around2_calcula(((org.aspectj.runtime.internal.AroundClosure)(null)),
                                checkTimeToProcess.aspectInstance);
}

public final int around2_calcula(final org.aspectj.runtime.internal.AroundClosure
                                       ajc$closure, final checkTimeToProcess this_){
   while (!this_.timeToProcess()){
     //sleep();
   }

   System.out.println("Comenzando calculos..");
   int result = this.dispatch2_calcula();
   System.out.println("Calculos terminados!");

   return result;
}

final int dispatch2_calcula(){
   int result = 0;
   for (int i = 0; i < this.exponente; i++)
      result = result + this.base * this.base;
   return result;
}
```

El método calcula ha dado resultado en tres métodos. El original `calcula` llama a otro `around2_calcula`, donde se encuentra el código definido en el aviso y en el lugar donde se hace la llamada a `proceed` en el código original se ha sustituido por `this.dispatch2_calcula`. Esta función contiene el código original de la función

tal como está escrito en la clase Potencia. Examinado los ficheros nos encontramos bastante más código, que sirve fundamentalmente para la captura de información contextual.

Con objeto de poder comparar literalmente el código original de la clase Potencia y el código generado por el compilador `ajc` una vez que ha entrelazado el código de los dos aspectos `Logger` y `checkTimeToProcess` puede verse la tabla siguiente donde a la izquierda se muestra el código original y a la derecha el código compilado.

package math;	package math; import checkTimeToProcess; import Logger;
public class Potencia{	public class Potencia { static org.aspectj.runtime.reflect.Factory ajc$JPF; private static org.aspectj.lang.JoinPoint.StaticPart getBase$ajcjp1; private static org.aspectj.lang.JoinPoint.StaticPart getExponente$ajcjp2; private static org.aspectj.lang.JoinPoint.StaticPart setBase$ajcjp3; private static org.aspectj.lang.JoinPoint.StaticPart setExponente$ajcjp4;
int base; int exponente;	int base; int exponente;

public int getBase(){
 	
return base;
}
	public int getBase() {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP
	(Potencia.getBase$ajcjp1, this, this, new java.lang.Object[] {});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	return this.base;
	}

public int getExponente(){
 	
return exponente
}
	public int getExponente() {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP
	(Potencia.getExponente$ajcjp2, this, this, new java.lang.Object[] {});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	return this.exponente;
	}

public void setBase(int dat){
 	
this.base=dat;
}
	public void setBase(int dat) {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP(
	Potencia.setBase$ajcjp3, this, this, new java.lang.Object[] {new Integer(dat)});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	this.base = dat;
	}

public void setExponente(int dat){
 	
this.exponente=dat;
}
	public void setExponente(int dat) {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP(
	Potencia.setExponente$ajcjp4, this, this, new java.lang.Object[] {new Integer(dat)});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	this.exponente = dat;
	}

public int calcula(){
 	
int result=0;
for (int i=0; i<exponente; i++)
result=result+base*base;
return result;
 	
}
	public int calcula() {
	return this.around2_calcula(((org.aspectj.runtime.internal.AroundClosure)(null)),
	checkTimeToProcess.aspectInstance);
	}
	
	final int dispatch2_calcula() {
	int result = 0;
	for (int i = 0; i < this.exponente; i++)
	result = result + this.base * this.base;
	return result;
	}
	
	public final int around2_calcula(final org.aspectj.runtime.internal.AroundClosure ajc$closure,
	final checkTimeToProcess this_) {
	while (!this_.timeToProcess()){/*sleep();*/}
	System.out.println("Comenzando calculos...");
	int result = this.dispatch2_calcula();
	System.out.println("Calculos terminados!");
	return result;
	}

```
                                    public Potencia() {    super();  }

                                    static {
                                     Potencia.ajc$JPF = new org.aspectj.runtime.reflect.Factory("Potencia.java", Potencia.class);
                                     Potencia.getBase$ajcjp1 = Potencia.ajc$JPF.makeSJP("method-execution",
                                          Potencia.ajc$JPF.makeMethodSig("1-getBase-math.Potencia----int-"), 8, 2);
                                     Potencia.getExponente$ajcjp2 = Potencia.ajc$JPF.makeSJP("method-execution",
                                          Potencia.ajc$JPF.makeMethodSig("1-getExponente-math.Potencia----int-"), 9, 2);
                                     Potencia.setBase$ajcjp3 = Potencia.ajc$JPF.makeSJP("method-execution",
                                          Potencia.ajc$JPF.makeMethodSig("1-setBase-math.Potencia-int:-dat:--void-"), 11, 2);
                                     Potencia.setExponente$ajcjp4 = Potencia.ajc$JPF.makeSJP("method-execution",
                                       Potencia.ajc$JPF.makeMethodSig("1-setExponente-math.Potencia-int:-dat:--void-"), 12, 2);
}                                    }
                                    }
```

8 REFERENCIAS RECOMENDADAS

- Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John. Irwin, *Aspect-Oriented Programming*, Xerox Palo Alto Research Center, 1997.
- Ivan Kiselev, *Aspect-Oriented Programming with AspectJ*, Sams Publishing, 2003.
- *The AspectJ™ Programming Guide*, the AspectJ Team, Xerox Parc Corporation
- Gregor Kiczales, et al. *An Overview of AspectJ*. In Proceedings of the 5th European Conference on Object Oriented Programming (ECOOP), Springer. Budapest, Hungary.
- Erik Hilsdale, Jim Hugunin, Wes Isberg, Gregor Kiczales, Mik Kersten, *Aspect-Oriented Programming with AspectJ*, 2002.
- Ramnivas Laddad, *Introduction to AspectJ*, Manning Publications Co. 2003.
- S. Clarke y Robert J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In proceedings of ICSE 2001.
- P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. *N degrees of separation: Multi-dimensional separation of concerns*. In Proceedings of the 21st ICSE'99, May 1999.
- D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, December 1972.
- T. Elrad, R. E. Filman, and A. Bader. *Aspect-Oriented Programming*. Commun. ACM, 2001.
- Aspect Oriented Software Development. http://aosd.net
- Kiczales G., Hannemann J. Design pattern implementation in Java and AspectJ
- Distributed System Group, Trinity College Dublin. http://www.dsg.cs.tcd.ie/
- Howard Kim,. AspectC#: An AOSD implementation for C#. September, 2002.
- AspectC++ Homepage. *http://www.aspectc.org*

Guía rápida de referencia de AspectJ

Juan Manuel Nieto Moreno [*]
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT

Está es una guía de referencia rápida de la sintaxis de AspectJ correspondiente a la versión 1.0.6. Pretende dar una visión rápida de la sintaxis del lenguaje. Su inclusión aquí es para facilitar el acceso a la sintaxis, teniéndolo todo recogido en el mismo documento.

1. Declaración de aspects

aspect *A* { … }
define el aspecto *A*
privileged aspect *A* { … }
A puede acceder a los atributos privados.
aspect *A* **extends** *B* { … }
A extiende a *B* que es una clase o aspecto abstracto.
aspect *A* **implements** *B* { … }
A implementa a *B* que debe ses una interface.
aspect *A* **dominates** *(B || C)* { … }
Los avisos del aspecto *A* tienen mayor precedencia que los de los avisos de *B* o *C*.

Forma general:

[**privileged**] [*Modificadores*] **aspect** *Id*
[**extends** *Clase]* [**implements** *Lista de clases]* [**dominates** *Lista de aspectos]*
{ Cuerpo }

2. Declaración de puntos de corte

private pointcut *pc*() : *call(void Foo.m())* ;
Un punto de corte visible únicamente desde el tipo que se define.
pointcut *pc*(*int i*) : *set(int Foo.x) && args(i)* ;
Un punto de corte con visibilidad de paquete que captura un entero.
public abstract pointcut *pc*() ;
Un punto de corte abstracto que puede ser referenciado desde cualquier sitio.
abstract pointcut *pc*(*Object o*) ;
Un punto de corte abstracto visible solo en el paquete en el que se define. Los puntos de corte que lo implementen deben capturar un objeto de tipo Object.

Forma general:

abstract [*Modificadores*] **pointcut** *Id* (*Lista de parámetros*) ;

[Modificadores] **pointcut** Id (Lista de parámetros) : Pointcut ;

3. Declaración de avisos

before () : *get(int Foo.y)* { ... }
Se ejecuta antes de acceder al atributo int Foo.y
after () **returning** : *call(int Foo.m(int))* { ... }
Se ejecuta después de llamar a *int Foo.m(int)* cuando se ejecute sin problemas.

[*] Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

after () **returning** (*int x*) : *call(int Foo.m(int))* { ... }
Igual, pero dándole el nombre *x* al valor que se devuelve.
after () **throwing** : *call(int Foo.m(int))* { ... }
Se ejecuta después de llamar a *int Foo.m(int)* cuando termina de forma anómala devolviendo una excepción.
after () **throwing** (*NotFoundException e*) : *call(int Foo.m(int))* { ... }
Se ejecuta después de llamar a *int Foo.m(int)* cuando termina de forma anómala devolviendo la excepción *NotFoundException*. A la excepción se le da el nombre *e* en el cuerpo del aviso.
after () : *call(int Foo.m(int))* { ... }
Se ejecuta después de llamar a *int Foo.m(int)* no importa como termine.
before(*int i*) : *set(int Foo.x) && args(i)* { ... }
Se ejecuta antes de dar valor al atributo *int Foo.x*. El valor que se le va asignar está contenido en una variable *i* que puede utilizarse en el cuerpo del aviso.
before(*Object o*) : *set(* Foo.*) && args(o)* { ... }
Se ejecuta antes de asignar valor a alguno de los atributos de *Foo*. Los tipos primitivos se convierten a sus respectivos java (*int* a *Integer,* por ejemplo) y le da el nombre *o* en el cuerpo del aviso.
int **around** () : *call(int Foo.m(int))* { ... }
Se ejecuta este aviso en lugar de la función *int Foo.m(int)*.Devuelve un entero. En el cuerpo del aviso se llama a la función original usando **proceed**(), que debe tener los mismos parámetros que el aviso.
int **around** () **throws** *IOException* : *call(int Foo.m(int))* { ... }
igual, pero en el cuerpo del aviso se puede lanzar la excepción *IOException.*
Object **around** () : *call(int Foo.m(int))* { ... }
Igual, pero el valor de **proceed**() se convierte a un objeto del tipo *Integer*, y el cuerpo del aviso también debe devolver un *Integer* que se convertirá en un *int.*

Forma general:

[**strictfp**] *AdviceType* : *Pointcut* { *Cuerpo* }

donde *AdviceType* puede ser:

before (*Lista de parámetros*)
after (*Lista de parámetros*)
after (*Lista de parámetros*) **returning** [(*Parámetro*)]
after (*Lista de parámetros*) **throwing** [(*Parámetro*)]
Tipo **around** (*Lista de parámetros*) [**throws** *Lista de Excepciones*]

4. Variables especiales

thisJoinPoint
información contextual reflectiva sobre el punto de corte.
thisJoinPointStaticPart
el equivalente a **thisJoinPoint.getStaticPart()**, pero usando muchos menos recursos.
thisEnclosingJoinPointStaticPart
la parte estática del punto de corte que incluye a este.
proceed (*lista de parámetros*)
sólo disponible en el aviso **around**. La lista de parámetros debe tener el mismo número y tipo que la lista de parámetros que tiene el aviso.

5. Introducciones

int Foo . m (*int i*) { ... }
Introduce un método *int m(int)* en la clase *Foo*, con visibilidad de paquete. En el cuerpo del método la variable **this** se referirá a la instancia de *Foo*, no del aspecto.
private *int Foo* . *m* (*int i*) **throws** *IOException* { ... }
Introduce un método *int m(int)* que puede lanzar la excepción *IOException,* sólo visible dentro del aspecto. En el cuerpo del método la variable **this** se referirá a la instancia de *Foo*, no del aspecto.
abstract *int Foo* . *m* (*int i*) ;
Introduce un método abstracto *int m(int)* en la clase *Foo.*

Point . **new** (*int x, int y*) { ... }
Introduce un constructor en la clase *Point*. En el cuerpo del construcor, **this** se refiere al nuevo *Point*, no al aspecto.
private static *int Point . x* ;
Introduce un atributo estático *int* de nombre *x* en la clase *Point* visible sólo en el aspecto.
private *int Point . x = foo()* ;
Introduce un atributo no estático con valor inicial el resultado de *foo()*. En el inicializador, **this** se refiere a la instancia de *Foo*, no al aspecto.

Forma general:

[*Modificadores*] *Tipo Clase . Id* (*Lista de parámetros*) [**throws** *Lista de excepciones*]{*Cuerpo*}
abstract [*Modificadores*] *Tipo Clase . Id* (*Lista de parámetros*) [**throws** *Lista de excepciones*] ;
[*Modificadores*] *Clase* . new (*Lista de parámetros*) [**throws** *Lista de excepciones*] {*Cuerpo*}
[*Modificadores*] *Tipo Clase*. *Id* [= *Expresión*] ;

6. Cambios en la estructura estática

declare parents : *C* **extends** *D*;
Declara que la súper clase de *C* es *D*. Sólo válido si *D* está declarada extendiendo la súper clase original de *C*.
declare parents : C **implements** *I, J* ;
C implementa *I* y *J*.
declare warning : *set(* Point.*) && !within(Point)* : *"bad set"* ;
El compilador da un aviso con el mensaje *"bad set"* si encuentra que se le da valor a algún atributo de *Point* fuera del código de dicha clase.
declare error : *call(Singleton.new(..))* : *"bad construction"* ;
El compilador avisa de un error con el mensaje *"bad construction"* si encuentra una llamada a algún constructor de *Singleton.*
declare soft : *IOException || NotFoundException* : *execution(Foo.new(..))*;
Cualquier IOException o *NotFoundException* lanzada desde los constructors de *Foo* se envuelven en la excepción org.aspectoj.SoftException.

Forma general:

declare parents : *Clase* **extends** *Clase* ;
declare parents : *Clase* **implements** *Lista de clases*;
declare warning : *Pointcut* : *String* ;
declare error : *Pointcut* : *String* ;
declare soft : *Excepciones* : *Pointcut* ;

7. Primitivas en los puntos de corte

call (*void Foo.m(int)*)
Llamadas al método *void Foo.m(int).*
call (*Foo.new(..)*)
Llamadas a cualquier constructor de la clase *Foo.*
execution (* *Foo.*(..) throws IOException*)
Llamadas a cualquier método de la clase *Foo* que puedan lanzar la excepción *IOException.*
execution (*!public Foo .new(..)*)
La ejecución de cualquier constructor no público de la clase *Foo.*
initialization (*Foo.new(int)*)
Inicialización de cualquier objeto *Foo* que se haga con el constructor *Foo(int).*
staticinitialization(*Foo*)
Cuando el tipo *Foo* se inicializa, después de cargarse.
get (*int Point.x*)
Cuando se lee el atributo *int Point.x.*
set (*!private * Point.**)
Cuando se le asigna valor a cualquier atributo no privado de la clase *Point.*

handler (*IOException+*)
Cuando se captura una excepción *IOException* o un subtipo de ella en un bloque catch.
within (*com.bigboxco.**)
Puntos de enlace donde el código asociado esté definido en el paquete *com.bigboxco.*
withincode (*void Figure.move()*)
Puntos de enlace donde el código asociado esté definido en el método *void Figure.move().*
withincode (*com.bigboxco.*.new(..)*)
Puntos de enlace donde el código asociado esté definido en un constructor del paquete *com.bigoxco.*
cflow (*call(void Figure.move())*)
Puntos de enlace en el flujo de ejecución de las llamadas a *void Figure.move()* (incluyendo la propia llamada *void Figure.move()*).
cflowbelow (*call(void Figure.move())*)
Puntos de enlace en el flujo de ejecución de las llamadas a *void Figure.move()* (sin incluir la propia llamada *void Figure.move()*).
if (*Tracing.isEnabled()*)
Puntos de enlace donde *Tracing.isEnabled()* tiene el valor **true**. La expression lógica puede acceder únicamente miembros estáticos, variables del mismo punto de corte y **thisJoinPoint**.
this (*Point* || *Line*)
Puntos de enlace donde el objeto actual en ejecución es una instancia de *Point* o *Line.*
target (*java.io.InputPort*)
Puntos de enlace donde el objeto destino es una instancia de *java.io.InputPort.*
args (*java.io.InputPort, int*)
Puntos de enlace que tengan dos argumentos, el primero una instancia de *java.io.InputPort* y el segundo un *int.*
args (**, int*)
Puntos de enlace con dos argumentos, el segundo de los cuales es un *int.*
args (*short, .., short)*
Puntos de enlace con al menos dos argumentos, de los cuales, el primero y el último son *short*s.

Forma general:

call(*MethodPat*)	**get**(*FieldPat*)	**cflow**(*Pointcut*)
call(*ConstructorPat*)	**set**(*FieldPat*)	**cflowbelow**(*Pointcut*)
execution(*MethodPat*)	**handler**(*TypePat*)	**if**(*Expression*)
execution(*ConstructorPat*)	**within**(*TypePat*)	**this**(*TypePat* | *Var*)
initialization(*ConstructorPat*)	**withincode**(*MethodPat*)	**target**(*TypePat* | *Var*)
staticinitialization(*TypePat*)	**withincode**(*ConstructorPat*)	**args**(*TypePat* | *Var* , …)

donde:

MethodPat: [*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat* (*TypePat* , …)[**throws** *ThrowsPat*]
ConstructorPat: [*ModifiersPat*] [*TypePat* .] **new** (*TypePat* , …)[**throws** *ThrowsPat*]
FieldPat: [*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat*
TypePat:
IdPat [+] [**[]** …]
! *TypePat*
TypePat **&&** *TypePat*
TypePat || *TypePat*
(*TypePat)*