Programación por chequeo

Resumen

La programación orientada a objetos (POO) ha supuesto un avance importante en la ingeniería del software pero a medida que los sistemas se han hecho aun más complejos se ha identificado la necesidad realizar una separación de competencias (es decir, extraer de los métodos toda competencia que no les sea inherente), muy en especial las competencias de control de errores. La programación orientada a aspectos (POA) trata de conseguir esto pero es excesivamente compleja.

A este problema es al que se le busca dar solución, definiendo un enfoque de programación alternativo basado en la POO que permita la separación del control de errores y de la funcionalidad básica de los métodos al que se le a denominado "Programación por Chequeo" (PPCH); para dar soporte a las construcciones del nuevo enfoque se define una extensión al lenguaje de programación C#, una posible forma de implementarlas en C# estándar y se realiza la implementación parcial de un compilador que acepta estas nuevas construcciones.

Debido a la naturaleza y objetivos de este trabajo, este se define como un proyecto de I+D (investigación y desarrollo) en los que se requiere utilizar metodologías iterativas e incrementales. El desarrollo se dividió en dos grandes etapas: definición del nuevo enfoque e implementación de las nuevas construcciones; en la primera se emplea una adaptación al modelo en espiral y en la segunda prototipado evolutivo.

■Juan Luis Paz
Universidad Católica Andrés Bello,
Caracas, Venezuela
juanluispaz@gmail.com

■ William Torrealba
Universidad Católica Andrés Bello
Caracas, Venezuela
wtorrealba@gmail.com

Fecha de Recepción: 9 de mayo de 2008
Fecha de Aceptación: 1 de julio de 2008

Como resultado se ha añadido doce nuevas palabras reservadas y tres nuevas palabras contextuales al lenguaje de programación C# que dan soporte al nuevo enfoque, en contraste a las más de 30 palabras que añade el lenguaje de aspecto AspectJ a su lenguaje base; también se ha mostrado en un caso de estudio que de haber realizado la separación de competencias se habría ahorrado al menos el 18,25 % del código del sistema (en algunas partes de este el ahorro alcanza el 36,19 % del código).

Abstract

The object-oriented programming (OOP) was a major breakthrough in software engineering, however, as systems have become even more complex, the need make a separation of concerns has become apparent, especially those concerns involved with error control. The aspect-oriented programming (AOP) tries to figured out this but is too complicated.

The purpose here was to investigate solutions to this concern, by defining a programming alternative approach based on the OOP that separates error control and the basic functionality, called “Programación por Chequeo” (PPCH). This was to be achieved by defining an extension to the programming language C#, a possible way to implement them in C# standard, and a partial implementation of a compiler that accepts these new definitions.

Due to the nature and objectives of this work, this project was defined as research and development that required using iterative and incremental methodologies. The development was divided into two phases: definition of the new approach, and implementation of the new design. A spiral model of adaptation was used in the first phase and evolutionary prototyping in the second.

The results propose that twelve new reserved words and three new contextual words be added to the programming language C# that support this new approach, rather that the more than 30 words that adds aspect-oriented language AspectJ to their base language. A case study has shown that when error control is separated using this approach, code reductions of least 18.25% (and up to 36.19% in some parts) can be achieved.

1. Planteamiento del problema

En la historia de la ingeniería del software, se puede observar que los progresos más significativos se han obtenido gracias a la aplicación de uno de los principios fundamentales a la hora de resolver cualquier problema, incluso de la vida cotidiana, que no es más que la descomposición de un sistema complejo en partes que sean más fáciles de manejar, es decir, gracias a la aplicación del dicho popular conocido como “divide y vencerás”.

La programación orientada a objetos (POO) ha supuesto uno de los avances más importantes de los últimos años en la ingeniería del software para construir sistemas complejos utilizando el principio de descomposición, ya que el modelo de objetos subyacente se ajusta mejor a los problemas del dominio real que la descomposición funcional.

A medida que los sistemas se han hecho aun más complejos se ha encontrado otra problemática, hasta ahora en cada método se encuentra el control de errores y la funcionalidad básica del mismo mezclados, y esta mezcla hace difícil entender el código (y desvía la atención del programador del problema); por lo que se ha identificado la necesidad realizar una separación de competencias, es decir, separar el control de errores de la funcionalidad básica del método.

Hasta ahora hay tres enfoques que permiten de alguna forma la separación de competencias:

- la **programación orientada a aspectos** (POA), pero es excesivamente complicada, lo que conlleva una serie de problemas de magnitud tal que hacen dudar sobre la utilidad real de POA [21] y [2].
- la **programación por contrato**, la cual separa en cierta medida el control de errores pero no permite la reutilización, tampoco permite elaborar controles de errores más complejos

que el de comprobar si algo cumple con una serie de condiciones;

- hacerlo **manual**, creando clases con métodos a los que le delegan la competencia de realizar el control de errores, el inconveniente está en que se crean unas clases confusas dedicadas al control de errores.

Por lo que resulta necesario definir un nuevo enfoque de programación que permita la separación del control de errores de la funcionalidad básica del método, que facilite la reutilización de estos controles de errores, que se enmarque totalmente dentro de la programación orientada a objetos y que sea sencillo de utilizar ya que "la complejidad es el mayor enemigo de la calidad" [11].

2. Objetivo

Definir un enfoque de programación alternativo basado en la programación orientada a objetos que permita la separación del control de errores y de la funcionalidad básica de los métodos, así como la implementación parcial de un compilador que soporte el nuevo enfoque.

3. Metodología

Lo propuesto, por su naturaleza y objetivos, se define como un proyecto de investigación y desarrollo (I+D) y para afrontarlo se requiere de metodologías iterativas, flexibles, que permitan ir construyendo su producto de manera incremental y evolutiva.

Para afrontar la magnitud, el desarrollo de este trabajo se dividió en dos grandes etapas:

1. Definición del nuevo enfoque
2. Implementación de las nuevas construcciones

La metodología utilizada combina una adaptación al modelo de cascada con fases solapadas (permite que una fase se inicie sin que la anterior haya concluido) para el desarrollo general y un modelo iterativo incremental adaptado en cada una de las etapas antes listada.

La **primera etapa**, en la que se definen las construcciones del nuevo enfoque, es una etapa altamente creativa que genera una serie de documentos con las especificaciones del nuevo enfoque. Al ser esta una etapa creativa se necesita un modelo que permita ir construyendo su producto de manera iterativa e incremental, que permita ir mejorando y refinando las ideas surgidas en cada iteración a lo largo de todo el desarrollo; por lo que se utiliza una adaptación (se adapta ya que los productos de esta fase solo son documentos) al modelo en espiral que permite ir refinando el producto en cada iteración. En la primera iteración se inicia cada uno de los tres documentos productos de esta fase que iteración tras iteración es refinado, al final de cada iteración se decide si hay que volver a iterar.

La **segunda etapa**, en la que se busca crear un compilador que admita código C# ampliado con las nuevas construcciones (definidas en la etapa anterior) y genere código ejecutable. En esta etapa se decide si se va a crear un nuevo compilador o se va a modificar uno ya existente, también se debe seleccionar al principio de esta etapa cuales son las nuevas construcciones a implementar. La metodología utilizada en esta etapa es el prototipado evolutivo, modelo iterativo incremental que permite crear un compilador funcional que en cada iteración es ampliado; la elección del prototipado evolutivo se debe a que en esta etapa se debe afrontar el desarrollo de un producto con innovaciones importantes y el desconocimiento de cómo implementarlo (ya que no se trata de un sistema tradicional), esta decisión se ve reforzada si se modifica un compilador ya existente (vía que se efectivamente se ha tomado) ya que hay que afrontar el desconocimiento sobre el compilador electo. Durante cada iteración del prototipado evolutivo se incorpora una construcción al compilador de las ya seleccionadas para ser implementadas.

4. Definición del nuevo enfoque

Al tomar la idea de la programación orientada a aspectos de crear entidades especializadas para la separación de competencias de la funcionalidad básica del método, se tiene dos entidades: los aspectos (conservando la esencia expresada

en la programación orientada a aspectos) y los chequeadores, entidades especializadas en comprobar el cumplimiento o no de una o una serie de condiciones.

Al enmarcar estas entidades dentro de la programación por contrato, se tiene que el método mantiene la funcionalidad básica y sale de éste el control de errores, a una entidad similar a un aspecto, denominada contrato; también se tiene que cada método debe exponer de forma clara bajo cuáles contratos se rige.

Al descomponer el control de errores en objetos especializados en la comprobación de cierta situación en particular (por ejemplo: el número no puede ser negativo), da cómo resultado objetos con una alta cohesión, estos objetos se denominan chequeadores y representan las cláusulas de un contrato. Esta separación, adicionalmente, aumenta la cohesión del método ya que se concentra en la funcionalidad básica. El control de errores de un método se construye a partir de la combinación de esos chequeadores bajo la figura de un contrato, que es empleado en la cabecera del método, separándolo así de la funcionalidad básica.

5. Principios de diseño

Los principios en el diseño que se busca para todas las construcciones propuestas (listadas sin ningún orden en particular) son:

- Permitir la reutilización
- Utilización sencilla
- Capacidad de ser utilizadas en escenarios complejos
- Evitar la redundancia de código (por ejemplo: definir la cabecera de un método dos veces)
- Permitir la autodocumentación
- Construcciones legibles
- Evitar la posibilidad de introducir comportamientos sorpresivos
- Enmarcado dentro de la programación orientada a objetos

6. Aspectos

Tomando la esencia de los aspectos de la POA, en la programación por chequeo se redefine y simplifica, programáticamente hablando; la nueva definición es: “Unidad capaz de controlar las entradas y salidas de una unidad funcional”

Para dar soporte a la nueva definición de los aspectos se ha creado un nuevo miembro de clase denominado aspecto (no confundir con los aspectos de la POA).

Sintaxis general:

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

Los parámetros de *ingreso* son parámetros que existen al momento de ingresar a una unidad funcional (por ejemplo: en un método sus argumentos), los parámetros de *egreso* refiere a lo que retorna la unidad funcional (por ejemplo: en un método es el retorno de este). Tanto los parámetros de entrada como los de salida son lista de parámetros tal como se utilizan en los métodos.

Los aspectos poseen tres descriptores:

- El descriptor `pre` que 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` que 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` que 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 de un aspecto que muestra las entradas y salidas:

```
static class MostrarIOAspect {
  static aspect(object arg):(object
retorno) {
      pre {
       Console.WriteLine(
          "Argumento: {0}", arg);
      }
      post {
        Console.WriteLine(
            "Argumento: {0} " +
            "Resultado: {1}",
            arg, retorno);
      }
      handler {
      cacth (Exception ex) {
        Console.WriteLine(
            "Argumento: {0} " +
            "Excepción: {1}",
            arg, ex);
            throw;
            }
          } // fin aspect
       }
```

Para aplicar un aspecto sobre un método basta con colocar después del encabezado del método y antes de la apertura de llaves de implementación la palabra `aspect` seguido del nombre del contenedor del aspecto y a continuación los argumentos que el aspecto recibe. Ejemplo:

```
string AMayuscula(string arg)
      aspect MostrarIOAspect
             arg):(returned)
{
      return arg.ToUpper();
}
```

El valor retornado por el método se puede recuperar con la palabra `returned`.

También se pudo haber aplicado el aspecto sobre un bloque de código, utilizando la palabra `using` tal como se muestra a continuación:

```
void Main() {
      string texto = "Hola Mundo!";
      string enMayuscula;

      using aspect MostrarIO
                  (texto):(juntos) {
       enMatuscula = texto.ToUpper();
       }
     }
```

Si se desea aplicar más de un aspecto por vez, después de escribir la palabra `aspect` se puede listar la aplicación de los aspectos separados por coma, también se puede volver a escribir la palabra `aspect` y aplicar el siguiente aspecto. Ejemplo:

```
string MiMetodo()
      aspect A1Aspect():(),
             A2Aspect():()
      aspect A3Aspect():()
      { /* ... */ }
```

Para forzar la culminación normal de un descriptor en su implementación se utiliza la sentencia back. Ejemplo:

```
aspect(string s1, string s2):() {
      pre {
       if (s1 != s2)
        back;
         Console.WriteLine(
                      "Son iguales");
      }
    }
```

En el ejemplo anterior, si `s1` y `s2` no son iguales la llamada a escritura en la consola nunca será ejecutada, ya que el descriptor es culminado; tras la culminación del descriptor (bien sea utilizando la sentencia back o porque alcanzó la llave de cierre del descriptor) se ejecutan las instrucción sobre las cuales se aplica el aspecto.

7. Aspectos instanciados

En la iteración anterior se presentó cómo aplicar aspectos estáticos, pero los aspectos no serían totalmente compatible con la programación orientada a objeto si no fueran instanciables, por lo que para aplicar un aspecto que no posea el modificador `static` basta con escribir el nombre de la variable que referencia a la instancia en lugar de escribir el contenedor.

Ejemplo: El aspecto no estático (Ane)

```
class Ane {
      aspect(string s):() {
            pre { /* ... */ }
            post { /* ... */ }
```

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

se puede utilizar de la siguiente manera:

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

void HacerAlgo2(string s, Objeto o)
    aspect o.AspectoUsado(s):()
    {/*...*/}

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

También es posible declarar la variable que referenciará a la instancia del contenedor del aspecto e inicializarla en la misma aplicación, para ello después de la declaración de la variable se colocan los argumentos del aspecto y luego se hace la inicialización del mismo, tal como se muestra a continuación:

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

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

void HacerAlgo3(string s, Objeto o)
    aspect Ane b(s):() =
                o.GetAspectoUsado()
    {/*...*/}


void HacerAlgo4(string s, Objeto o)
    aspect Ane b(s):()=o.AspectoUsado
    {/*...*/}
```

Y para aquellos casos en los que no se desee crear una variable que referencie a la instancia del aspecto se puede crear la instancia del aspecto y luego colocar los argumentos que recibe, tal como se muestra a continuación:

```
void HacerAlgo(string s, Objeto o)
    aspect new Ane()(s):()
    {/*...*/}
```

8. Aspectos de inspección

Son aspectos que no pueden forzar la culminación normal de la unidad funcional sobre la cual se aplica. Es decir, las instrucciones sobre las que se aplica siempre se ejecutan (al menos que el descriptor `pre` inicie una excepción) y el descriptor `handler` no puede contener la excepción que dio origen a su ejecución (siempre se culmina el descriptor `handler` con el lanzamiento o relanzamiento de una excepción); dicho de otra manera, son aspectos que no están en capacidad de hacer culminar de forma normal la unidad funcional sobre la cual se aplica.

El tipo de los aspectos de inspección se puede colocar de manera explícita colocando después del paréntesis de cierre de los parámetros de egreso dos puntos y la palabra `noforce` (aplica para la declaración como para la aplicación); si se omite el tipo de aspecto se asume que se trata de un aspecto de inspección.

Ejemplo: Declaración explícita de un aspecto de inspección

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

Ejemplo: Aplicación de aspecto indicando explícitamente que es de inspección

```
string MiMetodo()
      aspect MiAspectoDeInspeccion
                        ():():noforce
      { /* ... */ }
```

9. Aspectos retornantes

Hay situaciones en las que se requiere que el aspecto tenga la capacidad de inducir el retorno sobre la unidad funcional sobre la cual se aplica, por ejemplo, un aspecto que tiene la competencia del manejo del control de errores en un ambiente en el cual los métodos retornan códigos de error (en lugar de emplear excepciones para indicar la

culminación anormal); en estas situaciones los aspectos de inspección no son los suficientemente poderosos.

Los aspectos retornantes son aspectos en los que las instrucciones sobre las que se aplica el aspecto no siempre se ejecuta (el descriptor `pre` puede provocar la culminación sin que el cuerpo sobre el cual se aplica se ejecute) y el descriptor `handler` puede contener la excepción que dio origen a su ejecución (a pesar de haberse iniciado una excepción el descriptor `handler` puede provocar la culminación normal); dicho de otra manera, son aspectos que tienen la capacidad de hacer culminar de forma normal la unidad funcional sobre la cual se aplica.

El tipo de los aspectos retornantes se debe indicar de manera explícita colocando después del paréntesis de cierre de los parámetros de egreso dos puntos y la palabra `force` (aplica para la declaración como para la aplicación).

Los aspectos retornantes tienen la limitante de que todos los parámetros de egreso deben poseer el modificar `out` ó `ref` (esto se debe a que deben estar en capacidad de darle un valor).

Para provocar la culminación normal de la unidad funcional sobre la cual se aplica basta con utilizar la sentencia `return` (previamente habiéndole asignado valor a los parámetros de egreso – de ser requerido –).

Ejemplo: Declaración de un aspecto retornante (`AR`), en el cual los errores se manejan con códigos de error

```
class MiAR {
 aspect (object entra)
        :(out int retorno):force {
 pre {
  if (entra == null) {
      retorno = -1;
      //Error, no puede ser nula
      return;
      }
    }
    post {
     retorno = 0; //Resultado correcto
    }
      handler {
        catch {
          retorno = -2; //Error, hay
excepción
      return;
       }
     }
  } // fin aspect
}
```

Ejemplo: Aplicación de un aspecto retornante

```
int MiMetodo(object o)
 aspect MiAR(o):(out
returned):force
{
  Console.WriteLine(o);
}
```

10. Chequeadores

Unidad diseñada para asumir la competencia de comprobar si sus argumentos cumplen con una serie de condiciones. Los chequeadores tienen dos formas de indicar si sus argumentos no cumplen con las condiciones exigidas: pueden retornar un booleano o pueden iniciar una excepción.

Al igual que los aspectos los chequeadores requieren de un contenedor, también se ha creado un nuevo miembro de clase denominado chequeador.

Sintaxis general:

checker(*parámetros***)**
{ /*...*/ }

Dentro de las llaves de implementación del chequeador se colocan las pruebas separadas por coma, cada prueba es del estilo (esta sintaxis evita tener que escribir una secuencia de if anidados):

- expresiónBooleana **else throw** excepción
- llamada a otro chequeador

Ejemplo:

```
static class EsParPositivoChecker
{
 public static checker(int n) {
  n >= 0 else throw
                  new NegativoExcep(),
  (n % 2)==0 else throw
                  new NoParExcep()
 }
}
```

Hay dos formas de utilizar un chequeador, en función del resultado que se desea obtener:

- Si se desea que inicie una excepción en caso de que falle una prueba, se llama se la siguiente manera (ejemplo):

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

- Si se sedea que retorne un booleano que indica si se superaron todas las pruebas, se llama de la siguiente manera (ejemplo):

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

Al igual que en los aspectos, existen chequeadores instanciados y su sintaxis de invocación es similar. Ejemplo:

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

Es posible utilizar la sintaxis abreviada de los chequeadores para las pruebas en cualquier otro lugar, para ello en lugar de invocar al chequeador se coloca dentro de paréntesis la implementación. Ejemplo:

```
void Ejemplo(int n) {
    check(
     n >= 0 else throw
               new NegativoExcep(),
     (n % 2) == 0 else throw
               new NoParExcep()
    );
 }
```

En el caso de checkis no se debe indicar la clausula else de las pruebas, solo se listan las expresiones booleanas separadas por coma. Ejemplo:

```
void Ejemplo(int n) {
    bool resultado = checkis(
     n >= 0,
     (n % 2) == 0
 );
 // ...
}
```

Hay situaciones en las que esta sintaxis abreviada para las pruebas no resulta ser lo suficientemente flexible para las operaciones que hay que hacer, para esos casos existe la implementación avanzada de chequeadores, que consiste en implementar por separado el caso check y el caso `checkis` como si cada uno se tratase de un método (solo que el retorno se hace con la sentencia `back`). Ejemplo:

```
static class EsParPositivoChecker {
 public static checker(int n) {
  check {
   if (n >= 0)
    if ( (n % 2) == 0 )
     back;
    else
     throw new NoParExcep();

    else
     throw new NegativoExcep();
    }
    checkis {
     if (n >= 0)
     if ( (n % 2)==0 )
      back true;
     else
      back false;

     else
      back fasle;
  }
 } // fin checker
}
```

11. Contratos

La programación por contrato separa del cuerpo del método la competencia de comprobar las entradas y salidas, esta separación también es posible de hacerla en la programación orientada a aspectos pero pierde capacidad (lo más resaltante es lo relacionado a la herencia).

Para la programación por chequeo se define una unidad especializada en la competencia de la comprobación de entradas y salidas denominada

contrato, los contratos son como aspectos especializados en la competencia ya mencionada, pero tienen la particularidad de que se heredan (si sobre un método es aplicado un contrato las redefiniciones del método se rigen por el contrato aun cuando en estas no se haya aplicado el contrato).

Al igual que los aspectos los contratos requieren de un contenedor, también se ha creado un nuevo miembro de clase denominado contrato.

Sintaxis general:

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

A diferencia de los aspectos, ninguno de los parámetros de ingreso y de egreso pueden recibir los modificadores `out` y `ref`, tampoco hay tipos de contratos.

Los contratos poseen dos descriptores:

- El descriptor **require** que indica las precondiciones o comprobaciones que se deben realizar antes de la ejecución de la primera instrucción del código sobre el cual se aplica el contrato.
- El descriptor **ensure** que indica las postcondiciones o comprobaciones que se deben realizar después de la ejecución de la última instrucción del código sobre el cual se aplica el contratos (solo se ejecuta en caso de salida normal, de haberse iniciado una excepción no se realizan las comprobaciones).

Tanto el descriptor `require` como el descriptor `ensure` son chequeadores, por lo que dentro de las llaves de implementación de estos descriptores se debe seguir la misma sintaxis utilizada para implementar los chequedores, también es posible utilizar la implementación avanzada de chequeadores en cada descriptor.

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 RCContract
{
 public static contract(
        double a):(double resultado)
 {
      require {
       a >= 0 else throw
                    new NegativoExcep()
}
ensure {
 a*a == resultado else throw
   new ResultadoErradoExcep(),
 resultado >= 0 else throw
   new ResultadoNegativoExcep()
   }
 } // fin contract
}
```

La aplicación de un contrato es similar a la aplicación de aspectos, pero en lugar de utilizar la palabra `aspect` se emplea la palabra `contract`. Si se desea aplicar aspectos y contratos, primero deben ir los contratos y luego aspectos. Ejemplo:

```
double CalcularRaizCuadrada(
  double n)
  contract RCContract(n):(returned)
  aspect LoggerAspect
                  (numero):(returned)
{ /* ... */ }
```

Al igual que en los aspectos, existen contratos instanciados y su sintaxis de invocación es similar. Ejemplo:

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

Ya que cada descriptor del contrato es un chequeador por si mismo, es posible invocarlos como tal, para ello basta con escribir después de la palabra `check` ó `checkis` el nombre del descriptor (`require` ó `ensure`) y luego realizar la invocación del contrato. Ejemplo:

```
  check require RCContract(4d):(0d);
  bool resultado = checkis require

     RCContract(4d):(0d);
```

A diferencia de los aspectos, los contratos se pueden aplicar sobre miembros abstractos y en interfaces, y los contratos son heredados por las reimplementaciones; es decir, aun cuando se cambie la implementación de un miembro, este se sigue rigiendo por el contrato que regía a su predecesor. Ejemplo:

```
interface ICalculadorRaizCuadrada
{
 double CalcularRaizCuadrada
        (double n)
         contract RCContract
                 (n):(returned);
}

class CalculadorRaizCuadrada
      : IcalculadorRaizCuadrada
{
 double CalcularRaizCuadrada(
                          double n)
 {
   // ...
 }
}
```

Aunque el método de la clase no diga explícitamente que se rige por el contrato RaizCuadradaContract este está sujeto al contrato ya que es parte de las cláusulas exigidas por el método de la interfaz que implementa.

12. Composición

Al aplicar el dicho “divide y vencerás” un problema se descompone en partes más simples pero la solución al problema está en la composición e interacción de las partes.

Al descomponer un aspecto complejo en aspectos más simples surge la necesidad de poderlos componer, para ello basta con aplicar los aspectos componentes al aspecto compuesto. Ejemplo:

```
static class AspectoComponente1 {
 static aspect
           (object o1):(object o2) {
  pre { /* ... */ }
  post { /* ... */ }
  handler { /* ... */ }
 }
}

static class AspectoComponente2 {
 static aspect
           (object o1):(object o2) {
  pre { /* ... */ }
  post { /* ... */ }
  handler { /* ... */ }
 }
}

static class AspectoCompuesto {
 static aspect
           (object o1):(object o2)
 aspect
      AspectoComponente1(o1):(o2)
 aspect
      AspectoComponente2(o1):(o2)
{
  pre { /* ... */ }
  post { /* ... */ }
  handler { /* ... */ }
 }
}
```

La funcionalidad del aspecto compuesto es el resultado de la combinación de la funcionalidad de los aspectos componentes y la propia. Es posible alterar el orden de composición indicando en que punto se debe colocar la funcionalidad propia, esto se logra haciendo una aplicación de aspecto pero en vez de invocar a uno se coloca la palabra `here`. Ejemplo:

```
static class AspectoCompuesto {
 static aspect
           (object o1):(object o2)
 aspect
      AspectoComponente1(o1):(o2)
 aspect
      here
 aspect
      AspectoComponente2(o1):(o2)
 {
 pre { /* ... */ }
 post { /* ... */ }
 handler { /* ... */ }
 }
}
```

Los contratos se pueden componer entre ellos de forma análoga a la composición de aspecto, aplicando los contratos componentes al contrato compuesto. Ejemplo:

```
static class ContratoComponente1 {
 static contract
           (object o1):(object o2) {
  require { /* ... */ }
  ensure { /* ... */ }
 }
}

static class ContratoComponente2 {
 static contract
           (object o1):(object o2) {
  require { /* ... */ }
  ensure { /* ... */ }
 }
}

static class ContratoCompuesto {
      static contract
           (object o1):(object o2)
 contract
      ContratoComponente1(o1):(o2)
 contract
      ContratoComponente2(o1):(o2)
 {
  require { /* ... */ }
  ensure { /* ... */ }
 }
}
```

13. Asignaciones preliminares

Hay situaciones en las que es deseable poder acceder al viejo valor de un argumento (por ejemplo: en el caso de un contrato que rige al método depositar en una cuenta, en donde es deseable poder comparar en las postcondiciones que el viejo saldo más el importe es igual al nuevo saldo), esto se resuelve creando un campo en el contenedor del aspecto o contrato al cual se le asigna el valor que se desea preservar tan pronto se conozca (esto se hace asignándole el valor al campo dentro del aspecto o contrato pero fuera de cualquiera de sus descriptores).

Ejemplo: Contrato que rige el método de depositar en una cuenta

```
struct RealizarDepositoContract {
 decimal balance_anterior;

 contract (
      decimal balance,
      decimal importe
    ):() {
 balance_anterior = balance;
 require {
      importe >= 0 else throw new

 ImporteNegativoExcep()
  }
  ensure {
   balance == balance_anterior
                + importe
    else throw
       new ResultadoErradoExcep()
  }
 } // fin contract
}
```

La clase cuenta con su método de depositar sería:

```
class Cuenta {
 decimal balance;

void Depositar(decimal importe)
 contract new
           RealizarDepositoContract()
                (balance, importe):()
 { /* ... */ }
 // ...
}
```

14. Aspectos y contratos anónimos y desmembrados

Con la finalidad de simplificar aun más la utilización de los contratos y los aspectos, se admite aplicar contratos y aspectos realizando su implementación in situ; permitiendo así realizar la separación de competencias sin tener que declarar de manera separada las competencias y la unidad funcional sobre la cual se aplica. Esto se consigue de dos maneras:

- Aspectos o contratos **anónimos**, en la cual se aplica el aspecto o contrato pero en vez de invocarlo se hace la implementación de este in situ. Para los aspectos y contratos anónimos se ha extendido las capacidades de las asignaciones preliminares permitiendo realizar la declaración de las variables que preservarán el valor. Ejemplo:

```
bool Depositar(decimal importe)
 contract {
  decimal balance _ anterior =
                      balance;
  require {
   importe>=0 else throw
        new ImporteNegativoExcep()
 }
 ensure {
  balance == balance _ anterior
             + importe
   else throw
        new ResultadoErradoExcep()
  }
 } // fin contract
{
 // ...
}
```

- Aspectos o contratos **desmembrados**, en la cual se permite aplicar e implementar el descriptor de aspecto o contrato deseado, para ello en lugar de anunciar la aplicación del aspecto o contrato se anuncia el descriptor y luego se realiza su implementación. Esta notación es la que más se acerca a la notación utilizada por los lenguajes que soportan programación por contrato. Ejemplo:

```
public long Factorial(long x)
 require {
  x >= 0 else throw
        new NegativoExcep()
 }
 ensure {
  returned>=x else throw
         new  ResultadoErradoExcep(),
  returned>=0 else throw
         new ResultadoNegativoExcep()
  }
{
 if (x <= 1)
  return 1;
 else
  return x * Factorial(x-1);
}
```

15. Propiedades envolventes

Hay situaciones en las que no se desea que un campo admita todos los valores que soporta, sino un subconjunto de ellos, estas situaciones en la programación por contrato se resolvían con invariantes de clase, pero para la programación por chequeo se presenta un mecanismo alternativo: propiedades envolventes combinadas con contratos que restringen los valores al subconjunto válido (como cláusula `require` del descriptor `set` de la propiedad).

Las propiedades envolventes permiten autocontener el campo que la soporta, restringiendo la visibilidad del campo a la propiedad que lo envuelve; por lo que si en algún otro lugar (dentro o fuera) de la clase o estructura se desea acceder al campo se debe hacer a través de la propiedad que lo contiene. La declaración e inicialización del campo autocontenido se hace dentro de la propiedad que lo envuelve pero fuera de cualquiera de sus descriptores.

Ejemplo: Clase cuenta bancaria donde hay que asegurar que el campo saldo nunca sea negativo

```
class CuentaBancaria
{
  public decimal Saldo {
  decimal _ saldo;
  get {
   return _ saldo;
}
set
 require {
  value >=0 else throw
      new   SaldoIncorecctoExcep()
   }
  {
  _ saldo = value;
  }
 }
}
```

En la programación por contrato las invariantes de clases se puede utilizar para expresar restricciones más complejas que asegurar que un campo contenga un subconjunto de los valores válidos para el tipo de dato (por ejemplo: que la combinación de valores existente en dos campos sea válida en el dominio del problema); para la mayoría de los casos con un buen diseño y una correcta utilización de la herencia y la descomposición en objetos simples, no se requieren de ese tipo de invariantes.

Por lo antes expuesto no se han incluido en la programación por chequeo las invariantes (estas

agregan una complejidad adicional al lenguaje que resulta útil en muy pocos casos y su objetivo se puede conseguir utilizando otras vías ya provistas por el lenguaje – en algunos casos sacrificando la separación de la invariante de la funcionalidad de la clase o estructura –), no obstante, en el trabajo especial de grado en el que se basa este documento se define una extensión a la programación por chequeo para dar soporte a las invariantes cuya presentación va más allá del alcance de este documento.

16. Principios del enfoque

A continuación se listan los principios de diseño que se deben tener en cuenta al utilizar la programación por chequeo:

- **Principio de no redundancia**: Bajo ninguna circunstancia se debe repetir en el cuerpo de un método (u otra unidad funcional) lo que hacen los aspectos y contratos que sobre este se aplican.
- **Principio de sencillez**: Se debe procurar hacer las cosas lo más sencillas posibles ("la complejidad es el mayor enemigo de la calidad"), para ello los métodos (u otras unidades funcionales) se deben limitar a hacer exclusivamente lo que a ellos les compete.
- **Principio de separación**: Siempre que sea posible, se debe extraer de los métodos (u otras unidades funcionales) toda competencia que no les sea inherente.
- **Principio de reutilización**: Se debe realizar la separación de competencias pensando y procurando su reutilización.

17. Resultados: el enfoque

Se ha diseñado un nuevo enfoque de programación que en cierta medida resulta ser un casamiento entre la programación por contrato y la programación orientada a aspectos, uniendo de esta manera estos dos enfoques sin aparente relación, tomando de ellas sus virtudes y deslastrándose de sus complicaciones, sin desenmarcarse de la programación orientada a objetos.

Frente a la programación tradicional orientada a objetos, la programación por chequeo permite:

- Realizar la separación de competencias de una forma fácil y sencilla
- Incorporar los conceptos más importantes de la programación por contrato y disfrutar de sus ventajas
- Mayor reutilización del código, como consecuencia de realizar la separación de competencias
- Mejorar notablemente la legibilidad del código, como consecuencia de realizar la separación de competencias
- Facilitar las tareas de mantenimiento, como consecuencia de haber aumentado la reutilización del código a raíz de haber realizado la separación de competencias
- Reducir notablemente la cantidad de líneas de códigos de un sistema, como consecuencia de haber aumentado la reutilización del código a raíz de haber realizado la separación de competencias
- Frente a la programación por contratos, la programación por chequeo permite:
- Reutilizar el control de errores
- Separar más clases se competencias, no solamente las ya soportadas por la programación por contratos: precondicones, postcondicones e invariantes
- Mayor reutilización del código, como consecuencia de realizar la separación de competencias
- Mejorar notablemente la legibilidad del código, como consecuencia de realizar la separación de competencias
- Facilitar las tareas de mantenimiento, como consecuencia de haber aumentado la reutilización del código a raíz de haber realizado la separación de competencias
- Reducir notablemente la cantidad de líneas de códigos de un sistema, como consecuencia de haber aumentado la reutilización del código a raíz de haber realizado la separación de competencias
- Frente a la programación orientada a aspectos, la programación por chequeo permite:

- Incorporar los conceptos más importantes de la programación por contrato y disfrutar de sus ventajas
- Solucionar los problemas relacionados con la herencia de la aplicación de aspectos
- Eliminar los problemas relacionados a los puntos de cortes y puntos de enlace
- Utilizar una sintaxis muchísimo más cercana a los lenguajes de programación orientados a objetos que la sintaxis utilizada en los lenguajes de aspectos, lo cual la hace más sencilla
- Utilizar una sintaxis muchísimo más sencilla y reducida que la provista por los lenguajes de aspectos; en lo aquí propuesto se añade doce palabras reservadas y tres palabras contextuales a C#, mientras que AspectJ (el lenguaje de aspectos más importante) añade más de 30 palabras significantes a su lenguaje base (java)
- Al estar en el mismo lugar la aplicación del aspecto y/o contrato y el bloque de código que lo recibe, permite saber rápidamente lo que allí pudiera ocurrir; a diferencia de los leguajes de aspectos en los cuales ambas cosas se encuentran separados, por lo que leyendo el bloque de código no se puede prever que sobre este se está aplicando un aspecto. Al tener ambas partes juntas mejora notablemente la autodocumentación del código
- Debido a la forma en que se aplican los aspectos y/o contratos es posible aplicarlos a partes de un método, sin tener que aplicárselo a la totalidad de este

Si se combina las construcciones provistas por la programación por chequeo con patrones de diseño se puede tener aplicaciones muy poderosas; por ejemplo, se podría tener una fábrica de aspectos que lea de un xml las acciones que va a ejecutar en `pre`, `post` y `handler`, permitiendo así ajustar su comportamiento en tiempo de ejecución sin tener que recompilar la aplicación.

18. Resultados: extención a C#

Se han diseñado las construcciones que dan soporte a la programación por chequeo como una extensión al lenguaje de programación C#, para ello se crearon:

- Doce nuevas palabras reservadas: `aspect`, `pre`, `post`, `handler`, `returned`, `back`, `checkis`, `check`, `checker`, `contract`, `require` y `ensure`
- Tres nuevas palabras contextuales: `force`, `noforce` y `here`
- Tres nuevos miembros de clases (y estructuras): aspectos, contratos y chequeadores que sirven para encapsular las competencias; cuya utilización se realiza en la cabecera del método, propiedad, bloque de código, etc. sobre el cual se desea aplicar la competencia.
- Dos nuevas instrucciones condicionales: `check` y `checkis`
- Una nueva instrucción de retorno: back
- Un nuevo concepto: la composición de competencias
- Un nuevo mecanismo de ocultación: las propiedades envolventes

Con esta extensión se da soporte a los conceptos de la programación por chequeo sin romper con la sintaxis tradicional del lenguaje C#, se aprovechan las construcciones ya provistas por este y se evita cualquier tipo de construcción sinónima, respetando siempre el espíritu y diseño original de C#.

Esta extensión no hace obsoleta ninguna de las construcciones del lenguaje, tampoco introduce incompatibilidades, por lo que preserva la compatibilidad con el código C# válido.

19. Resultados: documentos generados

Durante el desarrollo de lo que aquí se propone, se han generado una serie de documentos que explican con detalle el nuevo enfoque:

- Documento explicativo de las nuevas reglas y construcciones
- Documento con las extensiones a la gramática de C#
- Documento explicativo que indica como implementar las nuevas construcciones utilizando las construcciones ya provistas por C#

También se ha definido una extensión al nuevo enfoque para dar soporte a las invariantes, definida en la programación por contrato, pero no incluidas en el nuevo enfoque por la complejidad que añaden y el poco valor que aportan ante la mayoría de las situaciones, el nuevo enfoque provee de construcciones alternativas para las aplicaciones más útiles de las invariantes de clases.

20. Resultados: el compilador

Se ha construido un compilador (denominado cgmcs) que admite parte de las nuevas construcciones modificando el compilador ya existente para C# del proyecto Mono, las nuevas construcciones soportadas son:

- Implementación de aspectos de inspección.
- Implementación de contratos.
- Implementación de propiedades envolventes.
- Asegurar que el campo autocontenido sea únicamente utilizado por su propiedad envolvente.
- Aplicación de uno o varios contratos y/o aspectos de inspección sobre métodos, propiedades, constructores, constructores estáticos y redefinición de operadores.
- Aplicación de aspectos instanciados.
- Aplicación de contratos instanciados.
- Utilización de expresiones al momento de aplicar un aspecto o contrato, tanto para obtener la instancia de este como para indicarle sus argumentos.

Haber añadido todas estas construcciones al compilador gmcs no afecta a su comportamiento normal, por lo que el compilador sigue siendo compatible con las construcciones que este ya soportaba.

21. Caso de estudio

Como caso de estudio se ha seleccionado un software ya existente que ha sido desarrollada por completo sin realizar separación de competencias, en ella se ha buscado identificar algunas competencias y calcular el ahorro de código de haberse realizado la separación. El software seleccionado es SGEM (Sistema de Generación y Envío de Mensajes), que consta de 48.786 líneas de código escritas en C# distribuidas en 400 archivos.

Al estudiar el código se identificaron varias competencias, de las cuales se seleccionaron dos de ellas (las más significativas):

- 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 está abandonado un método.
- Competencia de tratamiento de errores de base de datos.

En el estudio se encontró que al separar las dos competencias seleccionadas mejoraba notablemente la legibilidad del código, facilitaba las tareas de mantenimiento y se reducían significativamente la cantidad de líneas de códigos del sistema. De haber realizado la separación de las competencias seleccionadas se habría ahorrado el 18,25 % del código total del sistema (en la implementación de acceso a base de datos el ahorro alcanza el 36,19 % del código).

22. Conclusiones

Se ha diseñado un nuevo enfoque de programación al que se le ha denominado “Programación por Chequeo” que permite la separación de competencias (muy especialmente las competencias de control de errores) y de la funcionalidad básica de los métodos basándose en la programación orientada a objetos. Este nuevo enfoque en cierta medida resulta ser un casamiento entre la programación por contrato y la programación orientada a aspectos, tomando de ellas sus virtudes y deslastrándose de sus complicaciones; uniendo así estos dos enfoques que no tienen un aparente punto en común.

Realizar la separación de competencias mejora notablemente la legibilidad del código, facilita las tareas de mantenimiento y se reduce significativamente la cantidad de líneas de códigos; tal como se muestra en el caso de estudio, en el cual aparte de haber obtenido los beneficios ya citados, de haberse

hecho la separación de competencias, se habría ahorrado el 18,25 % del código total del sistema; incluso, en la implementación de acceso a base de datos el ahorro alcanza el 36,19 % del código; hay que resaltar que para el caso de estudio solo se seleccionaron dos competencias de las varias existentes, por lo que el ahorro podría ser mayor.

Para dar soporte al nuevo enfoque se ha tomado como base el lenguaje de programación C# y se ha definido una extensión a este; que, sin romper con la compatibilidad con el código C# válido, le añade a este la capacidad de realizar la separación de competencias según lo dispuesto por la Programación por Chequeo, permitiendo así disfrutar de las ventajas de la separación de competencias en este lenguaje. También se definió una forma de implementar las nuevas construcciones utilizando C# estándar así como se implementó un compilador que acepta un subconjunto de estas.

Las capacidades de la Programación por Chequeo no se limitan a lo provisto por esta, ya que en ella se establece un marco flexible y extensible de trabajo de utilización genérica, y si se combina con patrones de diseño se puede tener aplicaciones muy poderosas; por ejemplo, se podría tener una fábrica de aspectos que lea de un xml las acciones que va a ejecutar en `pre`, post y `handler`, permitiendo así ajustar su comportamiento en tiempo de ejecución sin tener que recompilar la aplicación.

La elaboración de este trabajo ha sido ardua ya que se está trabajando con temas que son novedosos e inmaduros, en los que no existe mucha documentación escrita, por lo que necesario recurrir a documentos de conferencias y trabajos de grados. Con la Programación por Chequeo se busca dar un poco más de madurez a las ideas de la separación de competencias y hacerlas más factibles de utilizar en el proceso de desarrollo de software.

23. Recomendaciones: nuevas líneas de estudio

Siempre es preferible que un error se detecte en tiempo de compilación que en tiempo de ejecución, basándose en esta premisa, se podría dotar al compilador de la capacidad de identificar posibles violaciones a las cláusulas de los contratos. Véase el siguiente código:

```
static class EsParChecker
{
 public static checker(int numero)
 {
  (numero % 2)==0      else throw
      new   NumeroNoParException()
 }
}

static class Ejemplo
{
 static int GenerarNumeroPar()
  ensure {
      check EsParChecker(returned)
 }
{ /* ... */ }

static int GenerarOtroNumero()
{ /* ... */ }

static int Consumir(int numero)
  require {
      check EsParChecker(numero)
 }
{ /* ... */ }

static void Prueba()
{
 int a = GenerarNumeroPar();
 // a cumple con
 // EsNumeroParChecker,
 // sin importar su valor

 Consumir(a);
 // no habrá problemas,
 // a cumple los requisitos

 int b = GenerarOtroNumero();
 // b podría ser cualquier
 // número, no necesariamente
 // cumple con EsParChecker
 // (no lo garantiza, no está
 // en la clausula ensure
 // de GenerarOtroNumero() )

 Consumir(b);
 // esta llamada es un potencial
 // punto de fallo, ya que el
 // argumento no necesariamente
 // satisface las precondiciones
 // del método. El compilador
```

```
 // podría notificar la
 // posibilidad de que en este
 // punto ocurra un fallo.
 }
}
```

Basándose en los requerimientos y garantías de cada contrato, el compilador podría tratar al código como una máquina de estados (donde cada estado es representado por cláusulas de contratos – chequeadores –) y buscaría todos los cambios de estados que no sean seguros (aquellos en donde las garantías no satisfacen los requerimientos).

Un compilador que trabaje con este esquema permitiría identificar en tiempo de compilación un sin número de fallos que de otra manera solo serían detectables en tiempo de ejecución (en pruebas o en producción).

24. Recomendaciones: Complemento deseable

Dotar al lenguaje con la capacidad de tener referencias que no admitan nulo es un buen complemento para la programación por chequeo, ya que muchas de las precondiciones son restricciones que comprueban que un argumento no sea nulo.

Ejemplo: sin soporte a referencias no nulas

```
void GuardarEnDisco(object objeto)
      require {
      objeto != null else throw
        new ArgumentNullException()
 }
{ /* ... */ }
```

Al incluir el soporte a referencias no nulas el compilador puede detectar en tiempo de compilación el intento de asignar un valor nulo a un elemento que no lo admite.

Ejemplo: reescritura del ejemplo anterior utilizando la notación de Spec#

```
void GuardarEnDisco(object! objeto)
{ /* ... */ }
```

Al incluir soporte a referencias no nulas se reduce la necesidad de escribir precondiciones para este caso tan particular, común y repetitivo; simplificando aun más el código y reduciendo la posibilidad de introducir errores.

25. Recomendaciones: Estrategias de implementación

Si se desea construir o modificar un compilador para dar soporte la programación por chequeo, las siguientes recomendaciones que hay que tener en cuenta:

Para implementar adecuadamente las reglas y construcciones definidas en la programación por chequeo realizar transformaciones de código a nivel del preprocesador no es suficiente, se requiere una integración más profunda con el compilador, sobre todo al momento de la aplicación de los aspectos y contratos, en la que hay que decidir cuál es la sobrecarga a invocar.

Para dar soporte a la programación por chequeo no es necesario realizar modificaciones a la máquina virtual, se puede incorporar todas las nuevas reglas y construcciones al compilador y hacer que este las resuelva en tiempo de compilación; si embargo, si se desea utilizar varios lenguajes de programación sin que esto cause inconvenientes sería conveniente soportar las nuevas reglas y construcciones a nivel de la máquina virtual (en especial las relacionadas a la herencia de los contratos).

26. Referencias

[1] Aho, Alfred; Sethi, Ravi y Ullman, Jeffrey (1998). Compiladores. Principios, técnicas y herramientas Addison Wesley.

[2] Altman, Rubén y Cyment, Alan (2004). SetPoint: Un enfoque semántico parala resolución de pointcuts en AOP. Tesis de Licenciatura, Licenciatura en Computación, Universidad de Buenos Aires, Buenos Aires, Argentina

[3] Balagurusamy, E (2007). Programación orientada a objetos con C++. McGraw-Hill. 3ra. Ed.

[4] Casas, Sandra; Marcos, Claudia; Vanoli, Verónica; Reinaga, Héctor; Sierpe, Luis; Pryor, Jane y Saldivia, Claudio (2005). Administración de Conflictos entre Aspectos en AspectJ. Paper, Universidad Nacional de la Patagonia Austral, Unidad Académica Río Gallegos y, UNICEN, ISISTAN Research Institute, Argentina

Grupo de Gestión de la Tecnología. El ciclo de vida [en linea]. Universidad Politécnica de Madrid <http://www.getec. etsit.upm.es/docencia/ gproyectos/planificacion /cvida.htm> [Consulta: 22 de septiembre de 2007]

[5] Heileman, Gregory (1998). Estructuras de Datos, Algoritmos, y Programación Orientada a Objetos. McGraw-Hill.

[6] Joyanes, Luis y Fernandez, Matilde (2002). C#. Manual de programación. McGraw-Hill.

[7] Joyanes, Luis y Zahonero, Ignacio (2002). Programación en Java 2. Algoritmos, Estructuras de Datos y Programación Orientada a Objetos. McGraw-Hill.

[8] Joyanes, Luis (1998). Programación orientada a objetos. McGraw-Hill. 2da. Ed.

[9] Larman, Craig (2003). UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado. Prentice Hall. 2a. ed.

[10] Mckim, James (1996). Programming by Contract. IEEE Computer Society. Marzo de 1996

[11] Meyer, Bertrand (1999). Construcción de software orientado a objetos. Prentice Hall.

[12] Nieto, Juan (2003). Programación orientada a aspectos aplicada a tecnologías WEB. Proyecto Fin de Carrera, Ingeniería Informática, Universidad de Sevilla, Sevilla, España

[13] Plosh, Reinhold (1996). Tool Support for Design by Contract. IEEE Computer Society. Proceedings of TOOLS - 26 Conference, Santa Barbara 1998

[14] Pressman, Roger (2005). Ingeniería del software. Un enfoque práctico McGraw-Hill. 6ta. Ed.

[15] Quintero, Antonia (2000). Visión General de la Programación Orientada a Aspectos. Paper, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Sevilla, España

[16] Rancán, Claudio Jorge (2003). Gestión de configuración de productos software en etapa de desarrollo. Trabajo final, Especialidad en control y gestión de software, Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina

[17] Rossel, Gerardo y Manna, Andrea (2003). Diseño por Contratos: construyendo software confiable [En línea]. Revista Digital Universitaria. 01 octubre 2003, vol. 5 no. 5 <http://www.revista.unam.mx/vol.4/num5/ art11/art11.htm> [Consulta: 12 de octubre de 2006]

[18] Sarwar, Syed M.; Koretski, Robert y Sarwar, Syed M. (2003). El libro de LINUX. Addison Wesley.

[19] Schildt, Herbert (2003). C#. Manual de referencia. McGraw-Hill.

[20].- Torrealba, William (2003). Introducción a la teoría de traductores e intérpretes. Guía de clases, Ingeniería Informática, Universidad Católica Andrés Bello, Caracas, Venezuela

[21] Tourwé, Tom; Brichau, Johan y Gybels, Kris (2003). On the existence of the AOSD-Evolution Paradox. Workshop on Software-engineering Properties of Languages for Aspect Technologies. AOSD

[22] Wikipedia, La enciclopedia libre. Programación orientada a objetos [en linea]. <http://es.wikipedia.org/w/index.php?title=Programaci%C3%B3n_orientada_a_objetos&oldid=11105792> [Consulta: 7 de septiembre del 2007]

[23] Wikipedia, La enciclopedia libre. Programación Orientada a Aspectos [en linea]. <http://es.wikipedia.org/w/in-

dex.php?title=Programaci%C3%B3n_Orientada_a_Aspectos&oldid=10168934> [Consulta: 7 de septiembre del 2007]

[24] Wikipedia, The Free Encyclopedia. Research and development [en linea]. <http://en.wikipedia.org/w/index.php?title=Research_and _development&oldid=158726790> [Consulta: 22 de septiembre de 2007]