UNIVERSIDAD CATÓLICA ANDRÉS BELLO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

Programación por Chequeo

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

Prólogo

Un día me encontraba haciendo un proyecto para la cátedra de sistemas operativos II, había escrito ya toda la lógica del sistema, pero, al agregarle el control de errores y las instrucciones para que imprimiera en los logs de auditoría, la lógica quedó oculta tras una cantidad de líneas de código nuevas; mis métodos que originalmente eran muy legibles se habían vuelto oscuros y difíciles de entender, incluso para mí; me dije que esto no habría sido así de haber utilizado la programación orientada a aspectos (como complemento a la programación orientada a objetos), lástima que fuera tan compleja.

Cuando llegó el momento de introducir mi propuesta de trabajo especial de grado aún recordaba aquello, y me dije: por qué no crear un nuevo mecanismo que subsane aquél problema, ya que los que conozco no lo hacen por completo: un nuevo mecanismo que tome las virtudes de la programación orientada a aspectos y la programación por contratos, pero que sea muy sencillo de utilizar y coherente con la programación orientada a objetos.

En este documento presento mi respuesta a lo planteado, lo he dividido en dos partes: la primera que presenta el marco teórico y una explicación del nuevo enfoque, y la segunda (bajo la figura de apéndices) donde detallo las nuevas construcciones para C# y presento una forma de implementarlas.

Agradecimientos

A la profesora Elizabeth Arapé, por siempre procurar ser mucho más que una profesora,
para quien no tengo palabras con las que expresarle mi cariño y respeto,
y agradecerle por lo mucho que me ha enseñado

A William Torrealba, por su labor de tutoría en este trabajo especial de grado

A los profesores Carlo Magurno y Wilmer Pereira,
por ser los profesores de los que más he aprendido a lo largo de la carrera

A la profesora Lúcia Cardozo, por haberme dado una mano cuando más la necesitaba

A mi compañera Kristal Lucero, por su gran apoyo y amistad

A mi compañera Mónica Vidao, por su amistad a lo largo de toda la carrera

A la Universidad Católica Andrés Bello,
para la que no tengo palabras con que expresar mi cariño hacia ella

Dedicatoria

A mis padres, Juan Vicente e Isabel, por darme esta oportunidad

A mis hermanos, Sergio y Jesús, por su apoyo y compañía

A mi abuelo Herodoto y mi tío Gerardo, por saber comprender mi silencio

Y muy especialmente a mi abuela Rafaela, a quien extraño mucho,
que desde la grandeza del cielo me observa y guía

Contenido

Sinopsis

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.

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"; 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 especial de grado, 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.

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; 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).

Introducción

En la historia de la ingeniería del software, se puede observar que los progresos más significativos se han obtenido gracias a 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; pero a medida que los sistemas se han hecho aún 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 también de mantener); por lo que se ha identificado la necesidad realizar una separación de competencias, es decir, extraer de los métodos toda competencia que no les sea inherente.

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

- la **programación orientada a aspectos** (POA), pero es muy complicada, lo que conlleva una serie de problemas de magnitud tal que hacen dudar sobre la utilidad real de POA [Tourwé et al., 2003];
- 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.

Debido a que ninguna de las alternativas existentes satisface completamente las necesidades, resulta necesario definir un nuevo enfoque de programación que permita la separación de competencias sin perder facilidad de utilización, que se enmarque totalmente dentro de la programación orientada a objetos; y procure ser lo más sencillo posible ya que "la complejidad es el mayor enemigo de la calidad"[Meyer, 1999].

Este trabajo especial de grado busca dar solución a esto definiendo 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 al que se le a denominado "Programación por Chequeo"; para dar soporte a las construcciones del nuevo enfoque se define una extensión al lenguaje de programación C#, también se presenta una posible forma de implementarlas en C# estándar y se realiza la implementación parcial de un compilador que acepta estas nuevas construcciones.

Capítulo 1

Planteamiento del problema

Ante un problema dar una solución es el deber de un ingeniero, el problema que se presenta es: cómo realizar la separación de competencias de forma fácil y sencilla, muy especialmente la separación de la competencia del control de errores. En este capítulo se estudia el significado de este problema y se define este trabajo especial de grado que busca darle solución.

1.1 Descripción

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;
- 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, sea sencillo de utilizar y se enmarque totalmente dentro de la programación orientada a objetos.

1.2 Objetivo General

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.

1.3 Objetivos Específicos

- Definir el nuevo enfoque y las reglas aplicables en él
- Establecer para cada regla:
  1. Definición general
  2. Sintaxis de la regla para extender C#
  3. Una posible forma de implementar esa característica en C#: indicando la forma de transformar la regla en código entendible por un compilador de C# estándar
- Implementar de forma parcial un compilador que permita la posterior ejecución del código escrito bajo el nuevo enfoque

1.4 Alcance

- La versión de C# a usar es la establecida en el ECMA 334 3ra edición.
- No se van a implementar todos los elementos de C#, sólo las

construcciones básicas que definen al lenguaje, tales como: clases, métodos (con o sin parámetros simples, con o sin valor de retorno), variables, directivas de inclusión, declaración de variables, invocación de métodos y las instrucciones: `if`, `if – else`, `for`, `while`, `do – while`

- No se van a implementar todas las reglas del nuevo enfoque, sólo aquellas que muestren los conceptos básicos de éste; sin considerar las reglas involucradas a la herencia, ni a las construcciones anónimas, ni los mecanismos de chequeos dentro del código de los métodos.

1.5 Justificación

La **programación orientada a aspectos** (POA) busca realizar la separación de competencias a través de la figura de los aspectos, que lo logra separar bastante bien, y permite la reutilización de los mismos. Dos nuevos conceptos que introduce la POA [Altman y Cyment, 2004] son los puntos de enlace y puntos de corte: los primeros son puntos en la ejecución de un programa; los segundos, conjuntos de los primeros que permitirán luego especificar dónde corresponde aplicar un determinado aspecto. Estos dos conceptos (según las implementaciones actuales) conlleva una serie de problemas de magnitud tal que hacen dudar sobre la utilidad real de POA [Tourwé et al., 2003][Altman y Cyment, 2004]. Si a la problemática de estos dos conceptos se le suma una notación grande y compleja (utilizada por los lenguajes de aspectos) se tiene que la POA se vuelve excesivamente complejas de utilizar.

En la POA se separa de funcionalidad básica de los métodos toda competencia que no le sea inherente, y la responsabilidad de unir ambas cosas radica en los

puntos de corte, lo que genera un punto muy vulnerable en el sistema, es decir, si los puntos de corte están mal programados el sistema no funcionará adecuadamente; y resulta extremadamente fácil cometer errores allí.

Otro problema es la redundancia de la información que puede causar inconsistencia, esta redundancia la encontramos en que es necesario declarar de forma doble (como mínimo) la firma de los métodos a ser controlados por los aspectos: en la declaración del método y en el aspecto; además de tener también por duplicado el nombre completo de la clase a ser afectada por algún aspecto.

La POA tiene la capacidad de introducir puntos oscuros en el sistema si no se emplea con mucho cuidado, ya que ofrece cosas como poder invocar métodos inexistentes o el poder cambiar la línea de ejecución del programa (un aspecto puede interceptar la invocación de un método y decidir que este no será invocado y en su lugar invoca a otro sin que el programador sepa de forma clara que ello puede ocurrir), entre otras. Muchas de estas cosas rompen con las buenas prácticas de programación y atentan contra la legibilidad y predictibilidad del sistema, además de no permitir la autodocumentación.

Recientemente lenguajes de Aspectos como AspectJ han empezado a utilizar mecanismos para agregar metainformación a los métodos, como las anotaciones de java (atributos en C#), con esta notación se le agrega una anotación al método que indica la aplicación de un aspecto sobre este, evitando así tener que realizar los puntos de corte; aunque esto es una mejora sustancial aun queda el hecho de tener que lidiar con los puntos de enlace y una notación difícil de comprender, en cierta medida críptica, que no aprovecha los conocimientos previos. Los lenguajes de

aspectos suelen tener muchas palabras (AspectJ agrega más de 30 palabras con significado a las 50 palabras significantes de Java – Java es su lenguaje base –) y construcciones complejas que dificultan su utilización.

La **programación por contrato** también ha buscado la separación de competencias haciendo que el control de errores y la funcionalidad básica del método se programen en el mismo método pero en espacios separados, con esto logra separar ambas competencias, pero no permite la reutilización del control de errores, sin usar completamente el método (hay que aclarar aquí que se hace referencia al estilo de contratos en los lenguajes de programación: Eiffel, D y Spec# - lenguaje experimental de Microsoft -, no a la idea de programación por contrato como tal).

El problema en la programación por contrato está en:

1. la imposibilidad de reutilizar el control de errores;
2. es muy rígida, hay muchas cosas que no se pueden expresar en términos de contrato, la separación de competencias es más amplia que comprobar si algo cumple con una serie de condiciones, por ejemplo: en el control de errores hay una llamada para escribir en el registro de auditoría.

Los lenguajes de programación existentes basados en la programación por contrato (como Eiffel, D y Spec#) no logran poder expresar la totalidad de lo que la separación del control de errores puede llegar a ser.

Algunas personas, buscando la forma de separar el control de errores de la funcionalidad básica del método han **creado clases con métodos** a los que le delegan

la competencia del control de errores, al aplicar esta técnica en un método se tendría que al menos la primera línea (de ser necesario) invoca a otro método que se encarga de comprobar todos los argumentos, pasada la prueba de ese método se continúa con la funcionalidad básica del método y después podrían llamar a otro método para que compruebe el resultado.

El inconveniente con esta metodología de trabajo es que las clases a las que se les ha delegado la competencia del control de errores son confusas y muchas veces poco organizadas, ya que contienen una gran colección de métodos, siendo difícil clasificarlos en función de lo que comprueban o realizan, es decir, se convierten en clases con baja cohesión.

Debido a que ninguna de las alternativas existentes satisface completamente las necesidades existentes, resulta necesario definir un nuevo enfoque de programación que permita la separación de competencias y mejore la legibilidad del código sin perder facilidad de utilización, enmarcándose totalmente dentro de la programación orientada a objetos.

Capítulo 2

Marco teórico

A continuación se presentan y estudian los aspectos teóricos sobre los cuales se fundamenta este trabajo especial de grado.

2.1 Programación orientada a objetos

La **Programación Orientada a Objetos** (POO) [Wikipedia, 2007a] es un paradigma de programación que define los programas en términos de "clases de objetos", los objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos [Wikipedia, 2007a] expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas.

2.1.1 Objeto

Los **objetos** [Balagurusamy, 2007] son las entidades básicas en tiempo de

ejecución en un sistema orientado a objetos. Pueden representar a una persona, un lugar, una cuenta bancaria, una tabla de datos, o cualquier elemento que el programa tenga que manipular.

2.1.2 Clase

Una **clase** [Joyanes y Zahomero, 2002][Heileman, 1998] es una plantilla que define las características comunes a todos los objetos de un cierto tipo. Una clase [Heileman, 1998] contiene toda la información necesaria para construir ejemplares individuales de ella misma, estos ejemplares se conocen como objetos.

2.1.3 Encapsulamiento

El **encapsulamiento** o encapsulación [Joyanes, 1998] es la propiedad que permite asegurar que el contenido de la información de un objeto está oculta al mundo exterior. La encapsulación [Joyanes, 1998] (también conocido como ocultación de la información), es en esencia, el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales.

2.1.4 Herencia

La **herencia** [Joyanes y Zahomero, 2002] es el proceso mediante el cual un objeto adquiere las propiedades de otro objeto. Ello implica el concepto de clasificación jerárquica [Balagurusamy, 2007]. La herencia [Heileman, 1998] da la posibilidad de crear una nueva clase que sea una extensión o especialización de una clase existente (sin tener que volver a realizar las definiciones de sus miembros).

2.1.5 Polimorfismo

El **polimorfismo** [Joyanes, 1998] es la propiedad que indica, literalmente, la posibilidad de que una entidad tome muchas formas. En términos prácticos, el polimorfismo permite referirse a objetos de clases diferentes mediante el mismo elemento de programa y realizar la operación de diferentes formas, según sea el objeto que se referencia en el momento.

2.2 Programación por contratos

La **programación por contrato** [Meyer, 1999] fue introducida por Bertrand Meyer, el creador del lenguaje de programación Eiffel basándose en los trabajos de [Meyer, 1999] Bob Floyd, Tony Horae y Edsger Dijkstra; en ella se establece como contrato todas las condiciones que se deben cumplir para la correcta ejecución de un grupo de instrucciones.

2.2.1 Fórmulas de corrección

Sea `A` una cierta operación (por ejemplo, una instrucción o el cuerpo de una rutina). Una **fórmula de corrección** [Meyer, 1999] es una expresión de la forma

```
{P} A {Q}
```

y se interpreta como "Una ejecución de `A` que comience en un estado en el que se cumpla `P` terminará en un estado en el que se cumple `Q`"

Las formulas de corrección (también denominadas Tripletas de Hoare) son una notación matemática, no una construcción de programa [Meyer, 1999].

En `{P} A {Q}` (donde `A` denota una operación); `P` y `Q` son propiedades de las diferentes entidades implicadas. Estas propiedades también se denotan aserciones. De las dos aserciones `P` se denomina precondición (`require` – requiere) y `Q` postcondición (`ensure` – asegura). Un ejemplo básico sería: (suponiendo que x sea un número entero)

```
{ x >= 9 } x := x + 5 { x >= 13 }
```

Una **aserción** [Meyer, 1999] es una expresión que involucra algunas entidades de software y que establece una propiedad que dichas entidades deben satisfacer en ciertas etapas de la ejecución del software. Una típica aserción puede ser la que expresa que un entero es positivo o que una referencia no es nula.

El primer uso de las aserciones es la especificación semántica de las rutinas. Se puede especificar la tarea que lleva a cabo una rutina mediante dos aserciones asociadas a la rutina: una precondición y una postcondición. La precondición establece las propiedades que se tiene que cumplir cada vez que se llame a la rutina; la postcondición establece las propiedades que debe garantizar la rutina cuando retorne.

Ejemplo [Rossel y Manna, 2003]**:**

```
Modulo: raiz_cuadrada
  entradas: a: Real
  salidas: x: Real
  PRECONDICIONES
    {a>=0}
  INICIO
    //Sentencias del algoritmo
    return x;
  FIN
  POSTCONDICIONES
    {x*x == a} and {x>=0}
```

En lenguajes que soportan la programación por contratos (como Eiffel), se ha extendido las capacidades de las postcondiciones, permitiendo recuperar en estas el valor de un parámetro antes de que se ejecutara la rutina (anteponiendo la palabra `old` al nombre del parámetro).

2.2.2 Derechos y obligaciones

Definir una precondición y una postcondición es una forma de definir un contrato que liga a la rutina con quienes lo llaman [Meyer, 1999]. Al asociar la cláusula require (`pre`) y ensure (`post`) con una rutina r la clase le dice a sus clientes:

"Si usted me promete llamar a `r` con `pre` satisfecho entonces yo prometo entregar un estado final en el que `post` es satisfecho."

La característica más distintiva de los contratos, tal como ocurre en los asuntos humanos, es que un buen contrato entraña tanto obligaciones como beneficios para ambas partes (lo que es una obligación para uno se convierte en un beneficio para otro). Esto también es cierto para los contratos entre clases [Meyer, 1999]:

- **La precondición compromete al cliente:** define las condiciones bajo las cuales es legítima la llamada a una rutina. Es una obligación para el cliente y un beneficio para el proveedor.
- **La postcondición compromete a la clase:** define las condiciones que debe asegurar la rutina al retornar. Esto es un beneficio para el cliente y una obligación para el proveedor.

Los beneficios son, para el cliente, la garantía de que ciertas propiedades se

van a cumplir después de la llamada; y para el proveedor que se puede suponer que determinados presupuestos se van a cumplir cada vez que se llame a la rutina. Las obligaciones para los clientes son satisfacer los requisitos que establecen las precondiciones y para los proveedores cumplir con la tarea que establecen las postcondiciones.

Ejemplo [Rossel y Manna, 2003]**:**

	Obligaciones	Beneficios
Cliente	Entregar el paquete antes de las 17:00 con franqueo	El paquete estará en destino antes de las 10:00 del día siguiente
Mensajero	Entregar el paquete antes de las 10:00 del día siguiente	Dispone del paquete con tiempo y gana el valor del franqueo

2.2.3 Principio de no redundancia

"Bajo ninguna circunstancia el cuerpo de la rutina debe verificar el cumplimiento de la precondición de la rutina" [Meyer, 1999]

Esta regla es contraria a lo que abogan muchos de los libros de texto de ingeniería de software o de metodología de la programación, y que se conoce a menudo con el nombre de programación defensiva, en la cual se establece que para obtener software fiable hay que diseñar componentes que se protejan a sí mismos lo más posible. Es mejor comprobar demasiado, es lo que dicen esos enfoques, que demasiado poco; uno nunca es demasiado cuidadoso cuando tiene que tratar con extraños. Una comprobación redundante puede no ayudar, pero al menos no hace daño, dicen sus defensores [Meyer, 1999].

El diseño por contrato proviene de la observación opuesta [Meyer, 1999]: las

comprobaciones redundantes pueden hacer daño. Al extrapolar estas comprobaciones redundantes a las miles de rutinas de un sistema de tamaño medio comienzan a parecer un monstruo de complejidad inútil. Si se considera esta visión global, la sencillez se convierte en un criterio crucial ya que la complejidad es el mayor enemigo de la calidad [Meyer, 1999].

2.2.4 Invariantes de clase

Las precondiciones y las postcondiciones describen las propiedades de las rutinas individuales. También hay necesidades de expresar las propiedades globales de las instancias de una clase, que deben preservar todas las rutinas. Tales propiedades constituyen las invariantes (invariant) de la clase y capturan las propiedades semánticas más profundas y las restricciones de integridad que caracterizan a una clase [Meyer, 1999]. Un ejemplo de invariante para una clase que mantiene una cola es que la longitud de esta siempre debe ser mayor o igual a cero.

2.2.5 Violación de un contrato

Se quiera o no, y a pesar de todas las precauciones estáticas, algún suceso inesperado o no deseado ocurrirá más tarde o más temprano durante la ejecución de un sistema y generará una violación de un contrato trayendo como consecuencia el levantamiento de una excepción.

Excepción [Meyer, 1999]**:** Es un suceso en tiempo de ejecución que puede causar una rutina fracase.

Respecto al éxito y al fracaso de una rutina hay que decir [Meyer, 1999]:

- Una llamada a una rutina tiene éxito si termina su ejecución en un estado en el que satisface el contrato de la rutina. Fracasa si no tiene éxito.
- Un fracaso de una rutina causa una excepción en quien la llama.
- Una llamada a una rutina fracasa si y sólo si ocurre una excepción durante la ejecución de esta y la rutina no se puede recuperar de dicha excepción.

Como resumen se puede decir que un contrato es el conjunto de condiciones precisas que gobiernan las relaciones entre una clase proveedora y sus clientes. El contrato de una clase incluye los contratos individuales para las rutinas exportadas de la clase, representados mediante precondiciones y postcondiciones, y las propiedades globales de la clase, representadas por las invariantes de clase.

2.3 Programación orientada a aspectos

La **Programación Orientada a Aspectos** (POA) [Wikipedia, 2007b] es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de competencias (conceptos, incumbencias). Gracias a la POA se pueden capturar los diferentes competencias que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos . De esta forma se consigue razonar mejor sobre las competencias, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reutilizables [Wikipedia, 2007b].

Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA es utilizado para referirse a varias tecnologías relacionadas como [Wikipedia, 2007b] los métodos adaptivos (programación adaptativa), los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

El concepto de POA [Nieto, 2003] fue introducido por Gregor Kiczales y su grupo, aunque el equipo Demeter 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. 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 [Quintero, 2000][Nieto, 2003]:

> *"Un aspecto es una unidad que se define en términos de información parcial de otras unidades".*

Actualmente es más apropiado hablar de la siguiente definición de Gregor Kiczales [Quintero, 2000][Nieto, 2003]:

> *"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".*

Se puede diferenciar entre un componente y un aspecto viendo al primero como aquella propiedad que se puede encapsular claramente en un procedimiento, mientras que un aspecto no se puede encapsular en un procedimiento con los lenguajes tradicionales ya que se diseminan por todo el código.

2.3.1 Fundamentos de la POA

La programación orientada a aspectos establece que [Quintero, 2000] en las clases se implementa la funcionalidad principal de una aplicación (como por ejemplo, la gestión de un almacén), mientras que con los aspectos se capturan conceptos técnicos tales como la persistencia, la gestión de errores, la sincronización o la comunicación de procesos.

Para tener un programa orientado a aspectos se necesitan definir los siguientes elementos [Quintero, 2000]:

- Un **lenguaje para definir la funcionalidad básica**. Este lenguaje se conoce como lenguaje base. Suele ser un lenguaje de propósito general, tal como C++ o Java. En general, se podrían utilizar también lenguajes no imperativos.
- **Uno o varios lenguajes de aspectos**. El lenguaje de aspectos define la forma de los aspectos (por ejemplo, los aspectos de AspectJ se programan de forma muy parecida a las clases).
- **Un tejedor de aspectos**. El tejedor se encargará de combinar los lenguajes. El proceso de mezcla se puede retrasar para hacerse en tiempo de ejecución, o hacerse en tiempo de compilación.

2.3.2 Objetivos de la POA

Entre los objetivos que se ha propuesto la POA [Quintero, 2000][Nieto, 2003] están principalmente el de separar conceptos y el de minimizar las dependencias entre ellos. De la consecución de estos objetivos se pueden obtener las siguientes ventajas [Quintero, 2000][Nieto, 2003]:

- Un código menos enmarañado, más natural y más reducido.
- Una mayor facilidad para razonar sobre las materias, ya que están separadas y tienen una dependencia mínima.
- Más facilidad para depurar y hacer modificaciones en el código.
- 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 reutilizable y que se puede acoplar y desacoplar cuando sea necesario.

2.3.3 Conceptos importantes

Puntos de enlace (Join points) [Nieto, 2003]: 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.

Puntos de corte (Pointcuts) [Nieto, 2003]: son constructores donde se especifican los puntos de enlace y se captura información referente al contexto. Se puede pensar en puntos de corte como las reglas de especificación y en los puntos de enlace como las situaciones que satisfacen dichas reglas.

Avisos (Advice) [Nieto, 2003]: 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. 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.

Introducciones (Introduction) [Nieto, 2003]: Si lo anterior era para el entrelazado dinámico, las introducciones se utilizan 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 utilizarlos tal como si hubieran sido definidos en esta.

Declaraciones en tiempo de compilación (Compile-Time declaration) [Nieto, 2003]: esta es otra forma de implementar técnicas de entrelazado estático. Permite añadir advertencias y errores para en tiempo de compilación detectar ciertos patrones de utilización que queramos advertir o prohibir (en el caso de que los identifiquemos como errores).

Aspectos (Aspect) [Nieto, 2003]: Estos son la unidad central de la POA. 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.

Para conocer más detalles de la POA y los lenguajes de aspectos se recomienda consultar el proyecto de fin de carrera de Juan Manuel Nieto Moreno [Nieto, 2003]

(en el Apéndice J se incluye un extracto de esta).

2.4 Traductores

Los **traductores** [Torrealba, 2003] son un tipo de programas cuya función es convertir el código de un lenguaje en otro. Por ejemplo un compilador, que traduce código fuente en código objeto. Existen distintos tipos de traductores, entre ellos destacan: compiladores, intérpretes, ensambladores y preprocesadores

2.4.1 Compiladores

Un **compilador** [Aho et al., 1998] es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente.

En la compilación hay dos partes [Aho et al., 1998]: análisis y síntesis. La parte de **análisis** divide al programa fuente en sus elementos componentes y crea una representación intermedia del programa fuente. La parte de **síntesis** construye el programa objeto deseado a partir de la representación intermedia.

En compilación, **el análisis consta de tres faces** [Aho et al., 1998]:

1. **Análisis lineal,** en el que la cadena de caracteres que constituyen el programa fuente se lee de izquierda a derecha y se agrupa en componentes léxicos, que son secuencias de caracteres que tienen un significado colectivo.

En un compilador, el análisis lineal se llama análisis léxico.

2. **Análisis jerárquico,** en el que los caracteres o los componentes léxicos se agrupan jerárquicamente en colecciones anidadas con un significado colectivo.

   El análisis jerárquico se denomina análisis sintáctico. Este implica agrupar los componentes léxicos del programa fuente en frases gramaticales que el compilador utiliza para sintetizar la salida.

3. **Análisis semántico,** en el que se realizan ciertas revisiones para asegurar que los componentes de un programa se ajustan de un modo significativo.

   La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación del código.

Compiladores de compiladores

Poco después de escribirse el primer compilador, aparecieron sistemas para ayudar en el proceso de escritura de compiladores. A menudo se hace referencia a estos sistemas como **compiladores de compiladores**, generadores de compiladores o sistemas generadores de traductores. En gran parte, se orientan en torno a un modelo particular del lenguaje, y son más adecuados para generar compiladores de lenguajes similares al del modelo [Aho et al., 1998].

2.4.2 Intérpretes

Los **intérpretes** [Torrealba, 2003] son programas que ejecutan instrucciones escritas en un lenguaje de alto nivel. Existen varias formas de ejecutar un programa

y una de ellas es a través de un intérprete.

Un intérprete traduce instrucciones de un lenguaje de alto nivel a una forma intermedia la cual es ejecutada. Por el contrario los compiladores traducen a instrucciones de alto nivel directamente a lenguaje maquina [Torrealba, 2003].

2.4.3 Ensambladores

Los **ensambladores** [Torrealba, 2003] son un tipo de traductor que convierte programas escritos en lenguaje ensamblador en programas escritos en código máquina.

2.4.4 Preprocesadores

Los **preprocesadores** [Aho et al., 1998] son programas que producen la entrada para un compilador, y pueden realizar las funciones siguientes:

1. **Procesamiento de macros:** un preprocesador puede permitir a un usuario definir macros, que son abreviaturas de construcciones más grandes.

2. **Inclusión de archivos:** un preprocesador puede insertar archivos de encabezamiento en el texto del programa.

3. **Preprocesadores "racionales":** estos preprocesadores enriquecen los lenguajes antiguos con recursos más modernos de flujo de control y estructuras de datos.

4. **Extensiones al lenguaje:** estos preprocesadores tratan de crear posibilidades al lenguaje que equivalen a macros incorporadas.

Capítulo 3

Marco metodológico

A continuación se presenta la metodología utilizada para el desarrollo de este trabajo especial de grado.

3.1 Modelo en cascada

El modelo en cascada [Pressman, 2005], algunas veces llamado el ciclo de vida clásico, sugiere un enfoque sistemático, secuencial hacia el desarrollo del software, que se inicia con la especificación de requerimientos del cliente y que continúa con la planeación, el modelado, la construcción y el despliegue para culminar en el soporte del software terminado. Es el más utilizado [GGT, 2007], siempre que es posible, precisamente por ser el más sencillo.

3.2 Prototipado evolutivo

Este modelo propone [Rancán, 2003] que se deben implementar en forma gradual aquellos requisitos y necesidades que vayan resultando claramente

interpretado, de forma tal que el prototipo evolucione hacia el sistema final. Se suele recurrir al prototipado evolutivo [GGT, 2007] cuando no se conoce exactamente cómo desarrollar un determinado producto o cuáles son las especificaciones de forma precisa.

3.3 Modelo en espiral

El modelo en espiral [Pressman, 2005] es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de la construcción de prototipos con los aspectos controlados y sistemáticos del modelo en cascada. Proporciona el material para el desarrollo rápido de versiones incrementales del software. Cuando se aplica el modelo en espiral [Pressman, 2005], el software se desarrolla en una serie de entregas evolutivas. Durante las primeras iteraciones, la entrega tal vez sea un documento del modelo o un prototipo. Durante las últimas iteraciones se producen versiones cada vez más completas del software desarrollado.

3.4 Proyectos de I+D

La investigación y desarrollo (I+D) se refiere [Wikipedia, 2007c] al trabajo creativo emprendido sobre una base sistemática para aumentar las reservas de conocimiento, incluyendo el conocimiento del hombre, cultura y sociedad, y la utilización de estas reservas de conocimiento para idear nuevas aplicaciones.

La I+D [GGT, 2007] es costosa por depender de personal muy cualificado, por realizarse de modo generalmente artesanal y por requerir muchas iteraciones (para hacer frente a incidencias), que multiplican la duración del proyecto. En los

proyectos de I+D para el desarrollo de productos o procesos nuevos o significativamente modificados [GGT, 2007] existe una fase de construcción, aunque normalmente se trata de la realización de un prototipo.

3.5 Metodología utilizada

Este trabajo especial de grado por su naturaleza y objetivos se define como un proyecto de I+D y para afrontarlo se requiere utilizar metodologías iterativas, flexibles, que permitan ir construyendo su producto de manera incremental y evolutiva.

Para afrontar la magnitud, el desarrollo de este trabajo especial de grado 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 etapa antes listada, tal como lo ilustra la siguiente figura: (ver Figura 3.1)

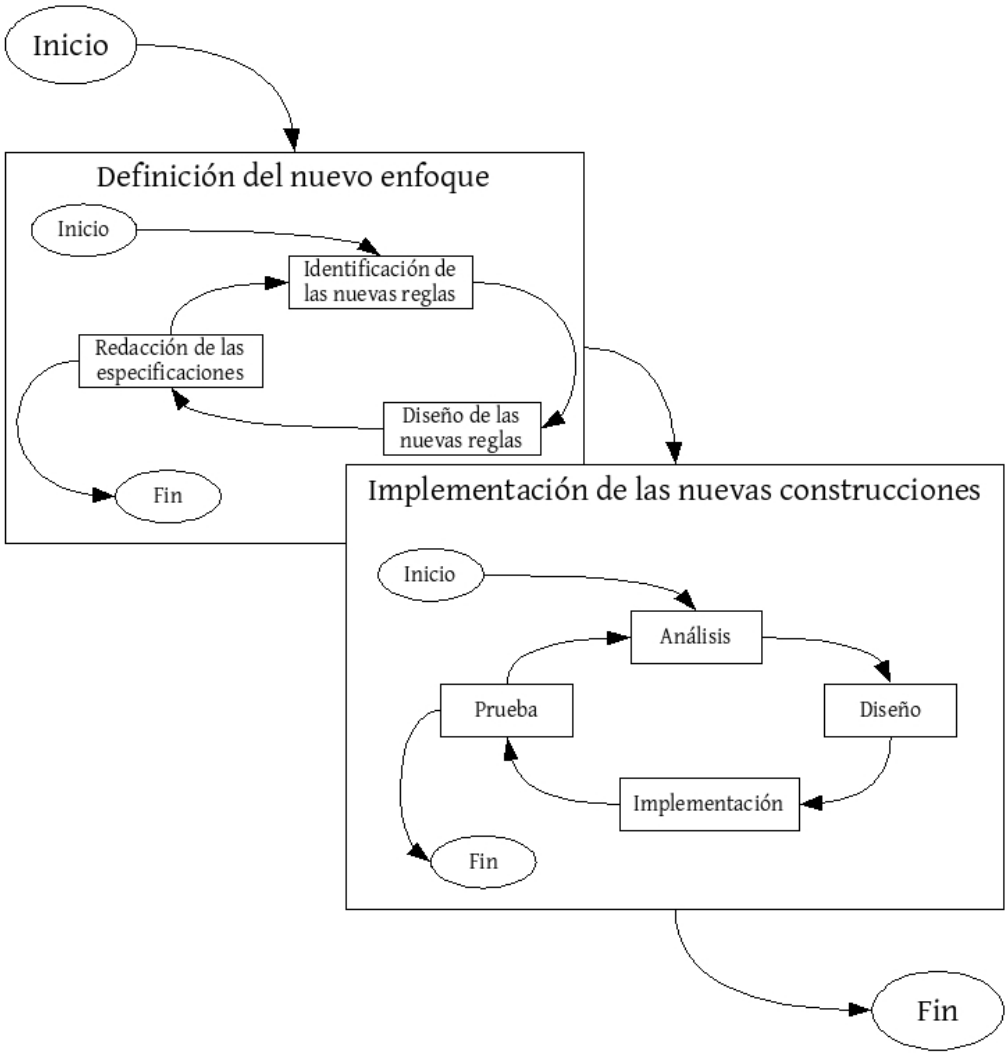

Figura 3.1: *Metodología utilizada*
Fuente: el autor

3.5.1 Etapa I: Definición del nuevo enfoque

En esta etapa 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.

Objetivo

Identificar, definir y documentar cada una de las nuevas reglas y construcciones del enfoque.

Productos

- Documento explicativo de las nuevas reglas y construcciones (incluido en el Apéndice A)
- Documento con las extensiones a la gramática de C# (incluido en el Apéndice C)
- Documento explicativo que indique como implementar las nuevas construcciones utilizando las construcciones ya provistas por C# (incluido en el Apéndice D)

3.5.2 Etapa II: Implementación de las nuevas construcciones

En esta etapa 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.

Objetivos

- Definir la estrategia de implementación: iniciar un nuevo compilador o modificar uno ya existente.
- Identificar las reglas y construcciones del nuevo enfoque a ser implementadas .
- Implementar un compilador que admita las reglas y construcciones seleccionadas.

Productos

Compilador que admita código C# ampliado con las nuevas construcciones (definidas en la etapa anterior) y genere código ejecutable.

Capítulo 4

Desarrollo

A continuación se muestra el desarrollo y evolución de este trabajo especial de grado durante su elaboración.

4.1 Etapa I: 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 clausulas 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.

Si se ve desde la óptica de la programación orientada a aspectos se tiene que los puntos de enlace desaparecen ya que estos están declarados en la cabecera de los métodos, pero se conserva la esencia de los aspectos; también se tiene unos aspectos especializados denominados contratos cuya competencia es comprobar las entradas y salidas de un método o unidad funcional.

Si se ve desde la óptica de la programación por contrato de tiene que los chequeadores son las cláusulas del contrato y la combinación de ellos hacen el contrato por el que se rige el método. Estos chequeadores se definen como un método especializado en la comprobación de un una situación, lo que permite definir cláusulas muy complejas sin aumentar la complejidad de su utilización.

4.1.1 Principios de diseño

A continuación se listan los principios en el diseño que se busca para todas las construcciones propuestas (listadas sin ningún orden en particular):

- 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
- Evitar la posibilidad de introducir comportamientos sorpresivos
- Construcciones legibles
- Enmarcado dentro de la programación orientada a objetos

4.1.2 Iteración 1: 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(parámetros de ingreso):(parámetros de 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: Aspecto que muestra las entradas y salidas por la consola

```
static class MostrarEntradaSalidaAspect {
  static aspect(params object[] args):(object retorno) {
    pre {
      Console.WriteLine(
        “Entrando con los siguientes argumentos {0}”,
        ListarArgumentos(args)
      );
    }

    post {
      Console.WriteLine(
        “Saliendo, se llamó con los siguientes argumentos {0}”+
        “ retornó el siguiente valor {1}”,
        ListarArgumentos(args), retorno.ToString();
      );
    }

    handler {
      cacth (Exception e) {
        Console.WriteLine(
          “Saliendo,se llamó con los siguientes argumentos {0}”
```

```
            + " culminó con error debido a la excepción {1}",
            ListarArgumentos(args), e.ToString();
          );
          throw;
        }
      }
    } // fin aspect

    private static string ListarArgumentos(object[] args) {
      string resultado = null;
      foreach (object obj in args)
        if ( resultado == null)
          resultado = obj.ToString();
        else
          resultado = resultado + ", " + obj.ToString();

      return resultado;
    }
}
```

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 Concatenar(string arg1, string arg2, string arg3)
  aspect MostrarEntradaSalidaAspect(arg1, arg2, arg3):(returned)
{
  return arg1 + arg2 + arg3;
}
```

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 s1 = "Hola ";
  string s2 = "Mundo";
  string s3 = "!";
  string juntos;

  Console.WriteLine("Antes de ejecutar el bloque de código");
```

```
  using aspect MostrarEntradaSalidaAspect(s1, s2, s3):(juntos) {
    juntos = s1 + s2 + s3;
  }

  Console.WriteLine("El resultado es {0}", juntos);
}
```

De haber ejecutado el código anterior la salida habría sido:

```
Antes de ejecutar el bloque de código

Entrando con los siguientes argumentos Hola , Mundo, !

Saliendo, se llamó con los siguientes argumentos Hola , Mundo, !
retornó el siguiente valor Hola Mundo!

El resultado es Hola Mundo!
```

Si se desea aplica 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 Aspecto1Aspect():(),Aspecto2Aspect():()
  aspect Aspecto3Aspect():()
{ /* ... */ }
```

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("Ambos argumentos son iguales");
  }
  post { /* ... */ }
  handler { /* ... */ }

}
```

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.

4.1.3 Iteración 2: 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

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

se puede utilizar de la siguiente manera:

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

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 coloca los argumentos del aspecto y luego se hace la inicialización del mismo, tal como se muestra a continuación:

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

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, OtroObjeto obj)
  aspect new AspectoNoEstatico()(s):()
{/*...*/}
```

4.1.4 Iteración 3: Aspectos de inspección y aspectos retornantes

Según su comportamiento, existen dos tipo de aspectos:

- **Aspectos de inspección:** no pueden forzar la culminación normal de la unidad funcional sobre la cual se aplica
- **Aspectos retornantes:** puede forzar la culminación normal de la unidad funcional sobre la cual se aplica

Aspectos de inspección

Son aspectos en los que las instrucciones sobre las que se aplica el aspecto siempre se ejecuta (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
{ /* ... */ }
```

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

```
class MiAspectoRetornate {
  aspect(object entrada):(out int retorno):force {
    pre {
```

```
        if (entrada == null) {
          retorno = -1; //Error: La entrada no puede ser nula
          return;
        }
      }
      post {
        retorno = 0; //Resultado correcto
      }
      handler {
        catch {
          retorno = -2; //Error: ha ocurrido una excepción
          return;
        }
      }
    } //fin aspect
}
```

Ejemplo: Aplicación de un aspecto retornante

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

Para conocer en detalle el comportamiento de los parámetros de entrada y salida, cómo se ven afectado tras el empleo de algún modificador y cómo utilizarlos véase el Apéndice A.

4.1.5 Iteración 4: 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 numero) {
    numero >= 0      else throw new NumeroNegativoException(),
    (numero % 2)==0  else throw new NumeroNoParException()
  }
}
```

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

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 numero) {
  bool resultado = checkis(
    numero >= 0,
    (numero % 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 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;
        }
      } //fin checker
    }
```

4.1.6 Iteración 5: 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(parámetros de ingreso):(parámetros de 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 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()
    }
  } //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 numero)
  contract RaizCuadradaContract(numero):(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 RaizCuadradaContract(4d):(0d);

bool resultado = checkis require RaizCuadradaContract(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 numero)
    contract RaizCuadradaContract(numero):(returned);
}

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

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.

4.1.7 Iteración 6: Composición y otras operaciones

Ante escenarios complejos la versatilidad y flexibilidad son requerimientos indispensables. En esta iteración se busca mejorar la interacción de las partes (contratos y aspectos) con su semejantes, también se busca atacar la situaciones en las que los cambios en el tiempo se ven involucradas (estas situaciones ya habían sido identificadas en la programación por contrato)

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` (para más detalles véase el Apéndice A). 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 { /* ... */ }
  }
}
```

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 ArgumentOutOfRangeException(
           “importe”, “importe no puede ser negativo”)
    }
    ensure {
      balance == balance_anterior + importe else throw
        new Exception(“El resultado del método es errado”)
    }
  } // fin contract
}
```

La clase cuenta con su método de depositar sería:

```
class Cuenta {
  decimal balance;

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

  // ...
}
```

4.1.8 Iteración 7: 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 ArgumentOutOfRangeException
        (“importe”, “importe no puede ser negativo”)
    }
    ensure {
      balance == balance_anterior + importe else throw
        new Exception(“El resultado del método es errado”)
    }
  } // 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 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);
}
```

4.1.9 Iteración 8: 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 SaldoIncorecctoException()
      }
    {
      _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 Apéndice E: “Extensión para el soporte de invariantes” se define una extensión a la programación por chequeo para dar soporte a las invariantes.

4.1.10 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” [Meyer, 1999]) , 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.

4.1.11 Extensión a la gramática de C#

Basándose en la gramática definida por la ECMA en ECMA-334 3ra edición en su Anexo A (presentada en este documento en el Apéndice B) y siguiendo la misma notación allí utilizada se presenta en el Apéndice C la extensión a la gramática de C# para dar soporte a las nuevas construcciones.

4.1.12 Forma de implementación

Una manera de implementar las nuevas construcciones es a través de métodos

subyacentes, y las aplicaciones no es más que la llamada a esos métodos subyacentes.

El el Apéndice D se presenta con detalle una forma de realizar la implementación de las nuevas construcciones utilizando las construcciones ya provistas por C#, basándose en la generación de métodos subyacentes.

4.2 Etapa II: Implementación de las nuevas construcciones

Para realizar la implementación de las nuevas construcciones hay dos estrategias posibles:

- Realizar un nuevo compilador
- Modificar un compilador ya existente

Para tomar la decisión se examinaron dos compiladores existentes de C#:

- Compilador de código abierto para C# de Microsoft, este compilador está escrito en C++
- Compilador para C# con genéricos (gmcs) de proyecto Mono (publicado como software libre), este compilador está escrito en C# y utilizan un compilador de compiladores (JAY) para generar la gramática.

La opción elegida es modificar el compilador para C# del proyecto Mono (versión 1.2.4), ya que el hecho de estar escrito en C# y el que utilice un compilador de compiladores simplifica notablemente el trabajo, también permite partir de una base ya existente y funcional en vez de tener que crear una nueva.

Para evitar complicaciones se decidió trabajar con el sistema operativo natural del proyecto Mono: Linux, la distribución elegida fue Ubuntu por ser una de

las más utilizadas.

En el Apéndice G se presenta algunos diagramas de clases del compilador para C# del proyecto Mono.

Para realizar la implementación se ha partido de la premisa que las construcciones deberán ser transformadas en sus traducciones en C# estándar tan pronto sean detectadas por el compilador, dicho de otra manera, la traducción del código se va a realizar en el mismo nivel en el que se interpreta las directivas del preprocesador, en el momento en el que se hace el análisis léxico.

Las construcciones seleccionadas a ser implementadas son:

- Implementación de propiedades envolventes
- Asegurar que el campo autocontenido sea únicamente utilizado por su propiedad envolvente
- Implementación de aspectos de inspección
- Aplicación de uno o varios aspecto de inspección sobre un método
- Aplicación de uno o varios aspecto de inspección utilizando la instrucción `using`
- Aplicación de uno o varios aspecto de inspección sobre propiedades, constructores, constructores estáticos y redefinición de operadores
- Implementación de contratos
- Aplicación de uno o varios contrato sobre un método
- Aplicación de uno o varios contratos utilizando la instrucción using
- Aplicación de uno o varios contratos sobre propiedades, constructores, constructores estáticos y redefinición de operadores

La implementación de las nuevas construcciones se realizó de manera iterativa: una construcción por iteración (en el mismo orden en que fueron listadas), esto permitió tener resultados corroborarles al finalizar cada iteración; la implementación de las construcciones se realizó en un total de diez iteraciones.

Gran parte de los cambios se realizaron en el documento con la gramática del compilador (documento de entrada de compilador de compiladores), pero se crearon un número de clases que dan apoyo en el proceso de traducción de las nuevas construcciones a código C# estándar, en el Apéndice H se presenta el diagrama de las nuevas clases.

Para poder asegurar que el campo autocontenido sea únicamente utilizado por su propiedad envolvente fue necesario hacer cambios más profundos, más allá del análisis léxico, este es el único caso en el cual no se da cumplimiento a la premisa para la implementación de las nuevas construcciones.

4.2.1 Limitaciones de la implementación

- El sistema no puede distinguir de manera correcta en todos los casos la sobrecarga de un aspecto o contrato a invocar, esto ocurre si la diferencia que existe entre ambas implementaciones es que varia el tipo de dato de alguno de los parámetros de egreso.
- El sistema mostrará un mensaje de error si se trata de utilizar un campo autocontenido fuera de su propiedad contenedora, pero solo si al campo se accede directamente, sin hacer desreferenciaciones, por lo que, si dentro del contenedor se accede al campo a través de una referencia

(como `this`) el sistema no mostrará el mensaje de error.

- No se implementaron las reglas asociadas a la creación de variables (en la aplicación del aspecto o contrato) que refenencien al contenedor de un aspecto o contrato.
- No se implemetó la regla que garantiza que al aplicar un aspecto o contrato utilizando el operador `new` la instancia del contenedor del aspecto o contrato se crea una sola vez, por lo que si se llega a aplicar un aspecto o contrato creando su instancia in situ, se va a crear la instancia cada vez que se llame a un descriptor del aspecto o contrato.
- El sistema exige que siempre que se declare un aspecto o contrato se coloque su implementación, no permite declaraciones abstractas.
- El sistema no soporta la implementación explícita de aspectos o contratos.
- No se soporta la implementación avanzada de descriptores de contratos.
- El compilador exige que al aplicar un aspecto o contrato se coloque un punto antes de la apertura de paréntesis para indicar los argumentos de ingreso; esta sintaxis se definió como opcional pero para el compilador implementado es obligatoria.

Capítulo 5

Resultados

A continuación se muestran los resultados obtenidos en este trabajo especial de grado.

5.1 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
- Mejorar notablemente la legibilidad del código
- Facilitar las tareas de mantenimiento
- Reducir notablemente la cantidad de líneas de códigos de un sistema

Frente a la programación por contratos, la programación por chequeo permite:

- Reutilizar el control de errores
- Separar toda clase se competencias, no solamente las precondicones, postcondicones e invariantes
- Mayor reutilización del código
- Mejorar notablemente la legibilidad del código
- Facilitar las tareas de mantenimiento
- Reducir notablemente la cantidad de líneas de códigos de un sistema

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

- Utilizar una sintaxis muchísimo más sencilla y reducida que la provista por los lenguajes de aspectos

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.

5.2 La extensión 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.

5.3 Documentos generados

Se han generado una serie de documentos que explican con detalle el nuevo enfoque:

- Documento explicativo de las nuevas reglas y construcciones (incluido en el Apéndice A)
- Documento con las extensiones a la gramática de C# (incluido en el Apéndice C)
- Documento explicativo que indica como implementar las nuevas construcciones utilizando las construcciones ya provistas por C# (incluido en el Apéndice D)

También se ha definido una extensión al nuevo enfoque para dar soporte a las invariantes (incluido en el Apéndice E), 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.

5.4 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

Adicionalmente, y aunque estuviera fuera de los objetivos, el compilador soporta:

- 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.

5.5 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 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).

En el Apéndice F se presenta el caso de estudio en detalle, estudiando el sistema en general y desglosándolo en cada una de sus capas.

Capítulo 6

Conclusiones

En este trabajo especial de grado 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 especial de grado 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.

CAPÍTULO 7

Recomendaciones

A continuación se presentan las recomendaciones relacionadas a la programación por chequeo en cuanto a su diseño y estrategia de implementación.

7.1 Nuevas líneas de estudios

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

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

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

    static int Consumir(int numero)
      require { check EsNumeroParChecker(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 EsNumeroParChecker (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).

7.2 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( “objeto”,
              “El objeto a guardar en disco no puede ser nulo”)
   }
{

   // ...
}
```

Al incluir el soporte a referencias no nulas el compilador puede detectar en tiempo de compilación el intento de 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.

7.3 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).

Bibliografía

[Aho et al., 1998] AHO, Alfred; SETHI, Ravi y ULLMAN, Jeffrey (1998). *Compiladores. Principios, técnicas y herramientas* Addison Wesley.

[Altman y Cyment, 2004] 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

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

[Casas et al., 2005] 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

[GGT, 2007] 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]

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

[Joyanes y Fernándes, 2002] JOYANES, Luis y FERNÁNDEZ, Matilde (2002). *C#. Manual de programación.* McGraw-Hill.

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

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

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

[McKim, 1996] MCKIM, James (1996). *Programming by Contract.* IEEE Computer Society. Marzo de 1996

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

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

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

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

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

[Rancán, 2003] 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

[Rossel y Manna, 2003] 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]

[Sarwar et al., 2003] SARWAR, Syed M.; KORETSKY, Robert y SARWAR, Syed M. (2003). *El libro de LINUX.* Addison Wesley.

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

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

[Tourwé et al., 2003] 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

[Wikipedia, 2007a] 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]

[Wikipedia, 2007b] Wikipedia, La enciclopedia libre. *Programación Orientada a Aspectos* [en linea]. <http://es.wikipedia.org/w/index.php?title=Programaci%C3%B3n_Orientada_a_Aspectos&oldid=10168934> [Consulta: 7 de septiembre del 2007]

[Wikipedia, 2007c] 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]