UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

UNA FORMA ORTODOXA PARA LAS CLASES EN C#

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

M A T E M Á T I C O

CON ORIENTACIÓN EN CIENCIAS DE LA

C O M P U T A C I Ó N

P R E S E N T A :

ROBERTO MANUEL JIMENO GÓMEZ

DIRECTOR DE TESIS:

M. EN C. JORGE LUIS ORTEGA ARJONA

FACULTAD DE CIENCIAS
UNAM

2004

VNIVERSIDAD NACIONAL
AVTENOMA DE
MEXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

Una forma Ortodoxa para las Clases en C#

realizado por Roberto Manuel Jimeno Gómez

con número de cuenta 08835570-2 , **quien cubrió los créditos de la carrera de:** Matemáticas con Orientación en Ciencias de la Computación.

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis
Propietario M. en C. Jorge Luis Ortega Arjona

Propietario Dr. Sergio Rajsbaum Godorezky

Propietario Dra. Hanna Oktaba

Suplente Dra. Amparo López Gaona

Suplente M. en C. María Guadalupe Elena Ibargüengoitia González

Consejo Departamental de Matemáticas

M. en C. Alejandro Bravo Mojica

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

Una Forma Ortodoxa para las Clases en C#

Roberto Jimeno

17 de noviembre de 2004

Manuel Jimeno Martínez
(1908 – 1958)
In memoriam

Índice general

1. Introducción

Ciencia es lo que conocemos suficientemente bien como para explicárselo a una computadora. Arte es cualquier otra cosa que hacemos.
– Donald Knuth

La Forma Canónica Ortodoxa (FCO) es una estructura de código propuesta por James Coplien para la declaración y composición de clases en C++. Está conformada por cuatro componentes: Un constructor por omisión, un constructor de copias, un destructor y la sobrecarga o redefinición del operador de asignación (i.e. el signo de *igual*: "="). La función de la FCO es determinar el comportamiento de las clases definidas por el programador, haciendo controlables a las instancias de las clases que usan la FCO.

C# es un lenguaje de programación creado por Anders Hejlsber y promovido por la compañía de software Microsoft[1] como un estándar aceptado por la Asociación Europea de Fabricantes de Computadoras (*European Computer Manufacturers Association* o simplemente *ECMA)* a finales del año 2001. C# es el lenguaje principal para desarrollar aplicaciones sobre una plataforma con un entorno independiente de ejecución y manejo de código denominada CLR (*Common Language Runtime*). Lo anterior implica que C# es el lenguaje a la vanguardia de los lenguajes destinados a usarse con el *framework* de .NET. Tiene influencia de varios otros lenguajes de programación y algunas recientes tecnologías asociadas a estos, como son la sintaxis de C++, la máquina virtual de Java, el uso de objetos de Smalltalk, el sistema de componentes de COM+, etc.

Sin embargo, debido a lo reciente de su creación, los programadores profesionales que usan C# aún son pocos y todavía no cuentan con suficiente experiencia para usarlo a su máximo potencial. Particularmente, y debido a esto último, aquellos programadores ya habituados al uso de ciertos patrones de software en otros lenguajes (como algunos modismos de C++ o Java) encuentran poco clara la expresión de tales patrones en este lenguaje

El objetivo del presente trabajo es proponer una forma para las clases de C# similar a la FCO en C++. La intención es mostrar que la FCO se puede aplicar con pocos cambios al ambiente de programación Orientado a Objetos de C#, así como analizar también las

[1] Además de otras compañías como Ximian (recientemente adquirida por la corporación Novell) y organizaciones como GNU (*Gnu's Not Unix*) y la FSF (*Free Software Foundation*), que también apoyan, promueven y desarrollan productos relacionados con .NET. Todos los ejemplos de este libro se desarrollan utilizando productos del proyecto Mono.

similitudes con el Objeto Canónico en Java, que es una fòrma similar a la FCO pero en el lenguaje Java.

1.1. Contexto

En el libro Advanced C++: Programming Styles and Idioms(COPJ94), Coplien propone varios modismos para C++, donde *modismo* se refiere a "un patrón utilizado como solución para un problema de programación en cierto lenguaje". En particular, la Forma Canónica Ortodoxa o simplemente *FCO (*en inglés *Orthodox Canonical Form* o simplemente *OCF)* se considera el modismo más básico y relevante, ya que indica o sugiere los elementos básicos que debe contener cualquier clase en C++. El uso de la FCO en C++ resulta conveniente, debido a que permite controlar la manera en que los objetos de las clases en C++ se comportan.

En general, la FCO necesita usarse en la declaración de una clase si(COPJ94):

- se desea permitir la asignación de objetos de esa clase, o si se desea pasar a sus instancias como parámetros *por valor* a una función, *y*
- las instancias contienen apuntadores a objetos sobre los que se realiza un conteo de referencias, o el destructor de la clase realiza un `delete` sobre alguna de las propiedades del objeto.

La FCO debe usarse para cualquier clase no trivial en un programa en C++, con el propósito de obtener uniformidad entre clases y para manejar la creciente complejidad de cada clase a lo largo de la evolución del programa.

1.2. Problema

El presente trabajo pretende desarrollar una expresión equivalente a la FCO pero en el lenguaje C#, lo cual implica ciertas consideraciones a partir de su expresión en C++. El objetivo es conservar la mayor parte de la intención y los resultados de la versión original de Coplien para C++(COPJ94).

1.3. Hipótesis

El propósito de este trabajo es responder a la siguiente pregunta como hipótesis:

¿Existe una expresión para la Forma Canónica Ortodoxa para las clases de C# que sea similar en forma (sintácticamente) y función (semánticamente) a la FCO propuesta originalmente por Coplien para las clases de C++?

En caso de que la respuesta anterior sea afirmativa, ¿cuál es esa forma, y de qué manera se modifican algunos de sus componentes?

1.4. Estructura de la tesis

Para responder a la pregunta planteada en la hipótesis se propone la siguiente estructura:

- El capítulo 2 define propiamente la FCO tanto en su forma como en su función.
- El capítulo 3 analiza la expresión de la FCO en el lenguaje C++ y el trabajo relacionado con la FCO en el lenguaje Java.
- El capítulo 4 propone la Forma Ortodoxa para las Clases de C# (FOCC#)
- El capítulo 5 muestra un ejemplo que se desarrolla basado en los conceptos propuestos en esta tesis.
- El capítulo 6 enuncia conclusiones.

1.5. Contribuciones

La principal contribución de esta tesis es la propuesta de un modismo equivalente a la FCO de C++ pero para el lenguaje C#. En otras palabras, la principal contribución es sugerir una forma de lograr que las clases de C# produzcan objetos cuyo comportamiento sea predecible y controlable.

Adicionalmente, el presente trabajo contribuye a fomentar el uso de modismos en la práctica de la programación. Ejemplo de ello son el uso del patrón Liberar (véase el apéndice B en la página 68) y la observación de que la estructura `using(){}` se comporta como un patrón (véase el apéndice C en la página 76).

Se muestra la importancia de definir un constructor por omisión, definir una política de copiado y conservar la relación entre las operaciones de comparación y de búsqueda en colecciones. Finalmente se describe un tratamiento al desperdicio de recursos.

2. Antecedentes

El buen juicio es resultado de la experiencia ...
La experiencia es el resultado del mal juicio.
– Frederick P. Brooks Jr.

El presente capítulo intenta introducir a los principales conceptos y al marco teórico de este trabajo. A continuación se enuncia el problema del software, que se refiere a la difícil situación del desarrollo de software a través del diseño y de la programación. En seguida, se mencionan los principales intentos para resolver, o al menos disminuir, tal problema. Los intentos en orden cronológico son: el desarrollo de lenguajes de programación, la Orientación a Objetos (OO) y los Patrones de Software. Posteriormente, se clasifica a los Patrones de Software y se definen los modismos, para así preparar el terreno para el siguiente capítulo, en el cual se analiza un modismo central llamado Forma Canónica Ortodoxa (FCO) de las clases en C++.

2.1. Introducción – La Crisis del Software

Desarrollar software no es trivial. El desarrollo de software es un campo mezclado de éxitos y fracasos. Varios estudios en el tema han mostrado que por cada tres nuevos proyectos de software a gran escala puestos en operación, uno es cancelado (WAYG94). Una revisión de la Oficina de Contabilidad Gubernamental de los EE UU sobre nueve proyectos de software para el Departamento de Defensa muestra que en 1979, aproximadamente 2 % de su presupuesto se gastó en software que se entregó y usó; sin embargo, aproximadamente 25 % del presupuesto se gastó en software que nunca se entregó, y aproximadamente 50 % se gastó en software que se entregó, pero nunca se utilizó (WINT96, pp. 106-107). Comúnmente, el proyecto de desarrollo de software promedio se excede 50 % del tiempo planeado. Mas aún, proyectos de mayor tamaño generalmente se retrasan todavía más. Y aproximadamente, tres cuartos de todos los sistemas grandes se consideran *fallas operativas*, es decir, o no se usan como se pretendía, o en absoluto no se usan (WAYG94).

Lo anterior son algunos ejemplos de lo que se conoce como la Crisis del Software (o *Software Crisis* en inglés), que describe la situación en el desarrollo de software en la cual los proyectos se entregan fuera de tiempo y de presupuesto, resultando además en software de pobre calidad y subutilizado.

Observando históricamente, se perciben algunas aproximaciones que han solucionado (aunque parcialmente) o pretendido solucionar algunos problemas en el desarrollo de

software. Estas aproximaciones son los Lenguajes de Programación, la Orientación a Objetos y los Patrones de Software.

2.2. Lenguajes de programación

Qué es un lenguaje de programación y cuál es su propósito no es fácil de responder. Si se desea incluir la mayor parte de las opiniones, entonces un lenguaje de programación es todo lo siguiente(STRB94):

- Una herramienta para darle instrucciones a las computadoras,
- un recurso para controlar dispositivos computarizados,
- un medio para comunicarse entre programadores,
- un vehículo para expresar diseños de alto nivel,
- una *notación* para algoritmos,
- una manera de expresar relaciones entre conceptos,
- una herramienta para experimentar

y la lista podría seguir.

Los Lenguajes de Programación (como Pascal o C) permiten al programador dejar de pensar en términos de *registros* y *direcciones* de memoria, para pensar en términos de *rutinas, funciones, apuntadores* y *estructuras de datos*. Seguramente el más poderoso empuje para la productividad, confiabilidad y simplicidad del software ha sido el progresivo uso de lenguajes de alto nivel para la programación. La mayor parte de los estudios acreditan a este desarrollo con al menos un factor de cinco en la productividad, y con ganancias concomitantes en confiabilidad, simplicidad y comprensibilidad (BROF75). De hecho, se considera que los programadores que trabajan con lenguajes de alto nivel logran mayor productividad y mejor calidad que aquéllos que trabajan con lenguajes de niveles más bajos. Lenguajes como C y Ada tienen el crédito de mejorar la productividad, confiabilidad, simplicidad y comprensibilidad más de 2.5 veces en el caso del primero y más de 4.5 veces en el caso del segundo, en comparación con lenguajes de bajo nivel, como ensamblador y lenguaje de máquina (BROF75).

Sin embargo, cuando se comienza a trabajar con un nuevo lenguaje de programación, se requiere de tiempo además de un esfuerzo importante por parte del programador para dejar de programar de la manera en que se encuentra acostumbrado, y comenzar a programar con el "estilo" del nuevo lenguaje. Dado que los lenguajes afectan la productividad y la calidad en el desarrollo del software en más de una forma, es de esperar que los programadores sean más productivos cuando usan lenguajes con los que están

familiarizados. Datos de la compañía TRW[1] muestran que de los programadores trabajando con un lenguaje que han usado por tres o más años son 30 % más productivos que los programadores con experiencia equivalente, pero son nuevos al lenguaje con el que trabajan (BOHB81). Un estudio en IBM encontró que los programadores con extensa experiencia en un lenguaje son tres veces más productivos que aquéllos con mínima experiencia (WALC77). Uno de los ejemplos más claros de esto sucede con los lenguajes de programación C y C++. Ya que el segundo es compatible con el primero, los programadores nuevos a C++ continuarán programando en C a menos de que durante algún tiempo se realice el esfuerzo de adaptarse al nuevo lenguaje, adoptando gradualmente las características que distinguen a C++ de C.

De manera similar, es de esperar que los programadores de nuevos lenguajes de programación como C#, y que tengan experiencia con C++, tardarán algún tiempo en adaptarse a las nuevas restricciones que este último impone, así como también les costará trabajo acostumbrarse a utilizar las nuevas características que C# proporciona. Como ejemplos, desearán poder sobrecargar el operador asignación ("=") mientras en contraste, no aprovecharán las propiedades *(properties)* ni los indizadores (*indexers*)[2].

2.3. Orientación a Objetos

El paradigma de Orientación a Objetos propone el uso de Tipos de Datos Abstractos relacionados jerárquicamente, para implementar sistemas como colecciones de subsistemas altamente independientes que interactúan entre sí a través de interfaces pequeñas y cuidadosamente diseñadas. En estos términos, un subsistema no accede los detalles de implementación de otros subsistemas. El manejo del *alcance* (o *visibilidad*) en esta forma es llamado *encapsulación*, que tiene base en el concepto de *ocultamiento de información* y produce diseños con menor acoplamiento entre sus distintos subsistemas (PARD72). Se dice que el bajo acoplamiento se traduce en gran facilidad para implementar los distintos sistemas de manera independiente. Además reduce el esfuerzo del mantenimiento, ya que las modificaciones a cualquier parte de la implementación son menos susceptibles de propagarse hacia otras partes del sistema, implicando que los cambios no se propagan por todo el sistema (LITT01, p. 17). De esta forma, los lenguajes de programación orientados a objetos permiten a los programadores desarrollar código eficientemente, y además, permiten que, una vez terminado el sistema, sea fácil de modificar y extender gracias a la utilización de técnicas como la *encapsulación* y la *herencia* (sencilla o múltiple).

Bajo el título de *Orientación a Objetos* están las ideas de *Tipos de Datos Abstractos*

[1] A mediados de la década del 70 ésta compañía producía satélites para departamento de defensa de los EE. UU. cuando se vió involucrada en un escándalo de espionaje donde la embajada de la URSS en la Ciudad de México tuvo una participación importante. Robert Lindsey convirtió la historia en un libro que posteriormente se llevó al cine estelarizada por Sean Penn bajo el título "The Falcon and the Snowman".

[2] Las propiedades son frecuentemente llamadas "campos inteligentes" porque de manera transparente añaden métodos públicos de acceso a campos declarados como privados o protegidos. Los indizadores son similares a las propiedades, pero estos permiten el acceso a campos del objeto a través de métodos que operan sobre índices; así un objeto puede operar como un arreglo.

y de *tipos jerárquicos*. Estas sirven para remover dificultades de la *expresión* del diseño, y no del diseño *en sí mismo*, lo cual implica que la Orientación a Objetos no consigue disminuir en forma definitiva la dificultad de diseñar software(BROF75, p. 189).

2.3.1. Conceptos clave

A continuación se presentan algunas definiciones y elementos relevantes de la Orientación a Objetos, comentando sus características más importantes.

2.3.1.1. Objetos y clases

Un objeto es una entidad identificable en un sistema orientado a objetos. Los objetos responden a *mensajes* realizando (*ejecutando*) una operación (*método*). Un objeto puede contener valores (de datos) así como referencias a otros objetos, los cuales en conjunto definen el estado del objeto. En consecuencia, un objeto tiene estado, comportamiento e identidad (BUSF96).

Las generalizaciones que describen conjuntos de objetos forman *clases*. Cada objeto tiene asociado un tipo. Cuando un grupo de objetos está asociado al mismo tipo se dice que pertenecen a la misma clase.

Tanto C# como C++, Java y otros lenguajes utilizan *clases* como descripciones de familias de objetos, y los objetos se crean como instancias (o ejemplares) de estas clases. Así, en una clase se definen los tipos de los atributos de objetos (i.e., las variables y constantes que representan su estado) y se implementan sus métodos (que son los procedimientos y funciones que modifican el estado). Sin embargo, no es sino hasta que usa o crea una instancia de la clase, que realmente se crea un objeto.

2.3.1.2. Abstracción, Encapsulación y Modularidad

La abstracción surge de un reconocimiento de similitudes entre ciertos objetos, situaciones o procesos en el mundo real, y de la decisión de concentrarse en estas similitudes e ignorar por el momento las diferencias. Sería impráctico intentar modelar un objeto como su contraparte real, con todas sus propiedades y comportamientos. Decidir el conjunto correcto de propiedades y comportamientos del objeto real que nos interesa modelar es obtener una abstracción del objeto real, es decir, la función de la abstracción es obtener las propiedades y comportamientos del objeto real que se consideran importantes para el modelo, y eliminar aquellas que no son preponderantes (ORTJ96, p. 59).

La encapsulación complementa a la abstracción ocultando al exterior del objeto aquello que no contribuye directamente a las características esenciales de éste. La encapsulación permite enfatizar la abstracción por medio de separar la interfaz y la implementación de un objeto, manteniendo oculta esta última (ORTJ96, p. 61).

La modularidad consiste en dividir un sistema en subsistemas más sencillos con los cuales es más confortable trabajar. Puede verse como un mecanismo para mejorar la flexibilidad y comprensibilidad de un sistema, y además puede ayudar a reducir el tiempo de desarrollo al permitir que distintos equipos de programadores trabajen de manera independiente en diferentes *módulos*. Un módulo se considera a cada uno de los subsistemas

del programa. Cada tarea a realizar por el programa debe corresponder a un módulo distinto dentro del programa (PARD72).

2.3.1.3. Tipos de datos jerárquicos

Es posible distinguir en la programación orientada a objetos algunas relaciones útiles entre clases de objetos. Considerar cada una o su combinación permiten realizar de cierta forma una jerarquización. Tales relaciones son principalmente: *herencia*, *agregación* y *uso* (ORTJ96, p. 64). En un mismo programa se puede tener más de una jerarquía de clases.

Herencia – *es-un*

La herencia expresa relaciones de tipo generalización/especialización, es decir, aquella relación en la que una clase comparte características de estructura o comportamiento definida en una o más clases.

La herencia es más que un mecanismo que simplifica la reutilización de código. La función principal de la herencia es expresar la compatibilidad de interfaces, es decir, el cumplimiento de una interfaz, también llamado *subtipificación*. En otras palabras, la herencia es más una técnica de *especificación* que una técnica de *implementación*. En los lenguajes orientados a objetos (como C++, Java y C#), las clases (no los objetos) heredan las propiedades de otras clases.

El polimorfismo es uno de los pilares de la programación orientada a objetos. El polimorfismo se basa en la herencia y el ligado dinámico, y permite en tiempo de ejecución la substitución de un objeto de un tipo dado, por cualquier otro objeto derivado de éste, es decir, alguno de sus subtipos o tipos derivados.

Se denomina *herencia múltiple* al tipo de herencia que permite que una clase derivada tenga más de una clase base. C++ permite la herencia múltiple, mientras que C# no la permite; sin embargo, C# permite múltiple herencia de interfaces y la *agregación* de objetos, lo cual conduce a obtener un resultado equivalente a la herencia múltiple(PRAP02).

Agregación – *tiene-un*

La agregación expresa relación del tipo todo/parte entre clases, esto es, por continencia o referencia entre dos clases de objetos(ORTJ96, p. 65). La diferencia con la herencia es que aquí ninguna de las clases en la relación es una especialización de la otra; en su lugar, sucede que una clase es parte de o contiene a otra (MURJ03, p. 14).

Uso – *usa-un*

La relación de uso es la más general de las tres. En este caso, las dos clases simplemente se comunican entre sí en algún punto del programa. Un objeto que *usa* otro objeto normalmente se comunica invocando a los métodos de este último (MURJ03, p. 14).

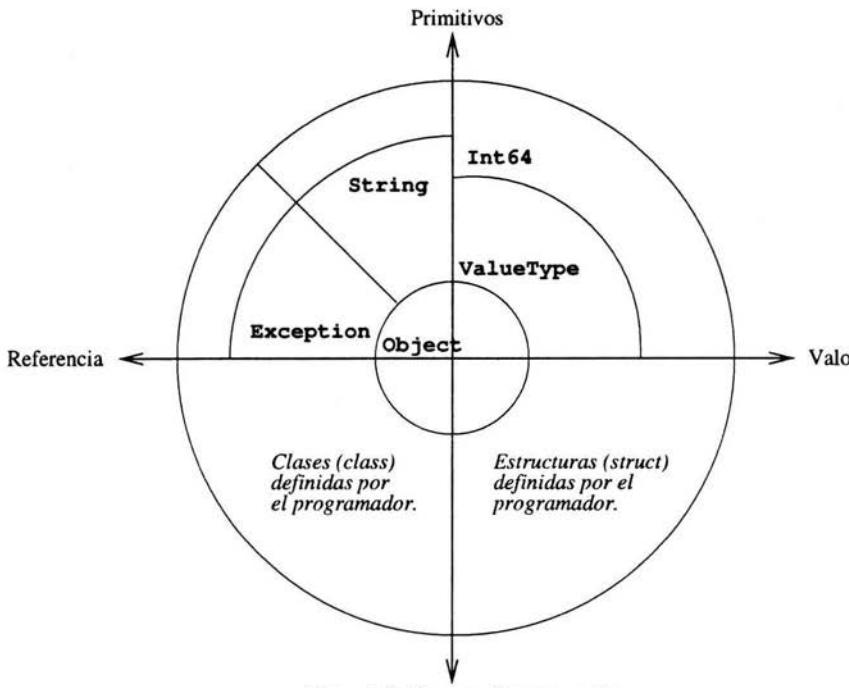
Sistema de tipos en C#

La base de la infraestructura de ejecución de C# es el sistema de tipos de datos de .NET (llamado CTS por sus siglas en inglés *Common Type System*). Los tipos de datos en este sistema pueden clasificarse de dos formas: a) tipos *primitivos* o tipos definidos por el programador y b) tipos valor o tipos referencia .

Con ambas clasificaciones se puede formar un cuadrante donde el eje horizontal indica si los datos son tipos referencia (acomodándolos hacia la izquierda) o tipos valor (acomodándolos hacia la derecha). Ligeramente a la izquierda del punto medio de este eje se encuentra el tipo `System.Object` que es el tipo referencia, es de donde se deriva `System.ValueType`, que es de donde se derivan todos los tipos valor.

De manera similar, el eje vertical clasifica a los datos entre tipos primitivos (acomodándolos en la parte superior), como tipos definidos por el programador (acomodándolos en la parte inferior del esquema). Nuevamente, `System.Object` se encuentra en la parte media de este eje ya que es el tipo primitivo más básico, del cual se derivan todos los tipos definidos por el programador.

En la siguiente ilustración se muestra donde queda clasificado el tipo `System.String`, que es un tipo primitivo y además es un tipo referencia. Se ha colocado cerca del centro del diagrama para representar su cercanía en la jerarquía de herencia con el tipo `System.Object`. En el diagrama, una clase se acerca o aleja más del origen (`System.Object`) dependiendo de su profundidad en la jerarquía de herencia. Así, `Int64` se se deriva de `ValueType`, que a su vez se deriva de `Object`.



En esta tesis nos referimos al cuadrante inferior izquierdo, es decir, a los tipos referencia definidos por el programador (i.e. `class`), de modo que no se menciona como implementar tipos valor (v.g. `struct`). Nuestra atención permanecerá centrada en la implementación por el programador de tipos referencia.

La relevancia de clasificar el sistema de tipos de .NET radica en que el comportamiento de las instancias de las clases en cada uno de los cuatro cuadrantes es diferente. Por ejemplo, la asignación se comporta diferente cuando se asignan tipos valor que cuando se asignan tipos referencia. De manera similar, el comportamiento de los tipos primitivos está predefinido y es inmutable, mientras que los tipos definidos por el programador pueden agregar comportamientos a los ya definidos en la clase `System.Object` (la clase base de todos los tipos definidos por el programador). No solo se puede agregar nuevos comportamientos a los definidos en `System.Object`; además se puede anular el comportamiento de sus métodos así como de algunas de sus operaciones.

2.3.1.4. Manejo de memoria, constructores y destructores

Cada clase define, ya sea de manera implícita o explícita, un conjunto de métodos dedicados a inicializar los objetos cuando éstos son creados, así como a liberar los recursos

usados por los objetos cuando éstos terminan su tiempo de vida. El conjunto de métodos responsables de la inicialización se llaman *constructores*; la responsabilidad de liberar los recursos recae en un único método al que se denomina *destructor*.

Probablemente, lo más importante que debe mencionarse respecto a los constructores es que proveen una "garantía" de cómo se inicializa un objeto. Si en una clase no se definen explícitamente los constructores de ésta, entonces no es claro cómo se inicializa una instancia de tal clase.

Por otro lado, durante su tiempo de vida un objeto utiliza recursos (v.g. memoria, descriptores de archivos, conexiones de red, etc). Cuando el objeto deja de existir, estos recursos deben ser liberados para que otros objetos puedan utilizarlos. Es aquí donde se utiliza el método destructor de la clase se vuelve importante. Cabe hacer notar aquí que el manejo de recursos se encuentra totalmente a cargo del programador de C++. En contraste, los recursos de C# son controlados principalmente por el ambiente de ejecución, lo cual marca una diferencia muy importante entre ambos lenguajes.

2.3.2. Lenguajes Orientados a Objetos

Anteriormente se menciona que los lenguajes, así como la OO son medios para reducir el problema del desarrollo de software. Los lenguajes orientados a objetos son la combinación de ambas soluciones, ya que incorporan lo descrito en la sección2.2 así como lo descrito en la sección2.3, de manera que sus beneficios se conjunten.

En algunos casos, los lenguajes orientados a objetos surgen a partir de lenguajes estructurados, extendiendo sus capacidades para expresar conceptos de OO. Un claro ejemplo de esto es C++, el cual es una extensión del lenguaje C que incorpora conceptos de OO (STRB91; STRB94). En otros casos, los lenguajes se crean desde el principio con la meta de incorporar los conceptos de OO.

Lenguajes estructurados

Los lenguajes estructurados organizan su código en bloques o grupos de instrucciones que tienen un nombre, y un único punto de entrada (i.e. un lugar único donde se comienza a ejecutar el código de ese bloque). Es común que a estos bloques se les llame procedimientos y funciones a los los procedimientos que regresan algún valor.

Lenguajes orientados a objetos

Los lenguajes OO organizan su código en clases. Las clases contienen conjuntos de datos protegidos a los que sólo se puede acceder a través de determinados métodos. Las clases se utilizan como prototipos para la creación de objetos en tiempo de ejecución. Los elementos declarados como "privados" son justamente los que están encapsulados.

Los miembros de una clase, ya sean métodos o atributos, tienen asociado un alcance o nivel de visibilidad. Los miembros *privados* son aquellos que sólo son visibles dentro de la misma clase. Los miembros declarados como *públicos* son visibles desde cualquier otro objeto (sin importar su clase). Hay, también, otros niveles de accesibilidad: Tanto en C++ como C# es posible declarar a los miembros como *protegidos* (`protected`) con

lo que el miembro sólo es accesible a otros miembros de su clase y de sus clases derivadas. Con C# es posible declarar a un miembro como interno (`internal`), con lo que éste solo es visible a otros objetos dentro del mismo módulo. Finalmente C++ permite "romper" la encapsulación, permitiendo la modificación de miembros privados por parte de objetos de clases declaradas como *amigas* (`friend`).

La programación orientada a objetos se basa en la programación estructurada. De hecho, los métodos de las clases se programan de manera estructurada. Los lenguajes orientados a objetos tienden a organizar "todo esto" en clases.

2.4. Patrones de Software.

Los Patrones de Software surgen dentro de la comunidad de programación orientada a objetos como *"descripciones de objetos y clases comunicantes que se adaptan para resolver un problema de diseño en un contexto particular"* (GAMM94, p. 3). Su finalidad es servir como base para diseñar sistemas de software. Los patrones tienen sus raíces en varias disciplinas, incluyendo muy notablemente el trabajo de Christopher Alexander de 1977 respecto a planeación urbana y arquitectura de edificios(COPJ01).

Existen muchas aproximaciones al concepto de patrón. Además de la definición inicial de la comunidad de programación orientada a objetos, un par más de ellas son:

- Un patrón es una solución probada a un problema recurrente en un contexto (GABR01).
- Cada patrón es una regla formada por tres partes que expresa una relación entre un cierto contexto, un problema, y una solución (GABR01).

Las dos definiciones anteriores son sencillas y lo suficientemente pequeñas como para ser fáciles de entender y manejar. Sin embargo, todavía no expresan claramente el concepto de patrón. Una definición más elaborada es:

Cada patrón es una regla de tres partes, que expresa una relación entre un contexto dado, un sistema de fuerzas que suceden repetidamente en ese contexto dado, y una cierta configuración de software, la cual permite a estas fuerzas resolverse consigo mismas (GABR01).

2.4.1. Formas de presentación de patrones (*templates*)

Los patrones de software se pueden presentar de distintas formas. Todas ellas sirven para expresar patrones y difieren entre sí sólamente respecto a la información o secciones que se consideran en ellas.

Entre las formas más comunes se encuentran la forma *Alejandriana*[3](ALEC77), la forma *GoF*(GAMM94), la forma *Portland*, la forma *Coplien* (COPJ00, pp. 12 - 14), la forma *POSA*(BUSF96), y otras.

[3](COPJ00, p. 2) sugiere usar *"Alexanderian"* en vez de *"Alexandrian"* para distinguirla de la forma poética Alejandrina. Aquí se usará el término *"Alejandriana"* para distinguirla.

Los patrones revisados en esta tesis se analizan utilizando una forma similar (casi un subconjunto) a la forma *POSA*, que se forma de secciones como se describe a continuación:

Nombre El nombre del patrón, junto con otros nombres bajo los que se le conozca, y un breve resumen.

Contexto La situación en la que el patrón se presenta.

Problema El problema hacia el cual el patrón está dirigido, incluyendo una discusión de las fuerzas asociadas a éste.

Solución Descripción de la solución en texto sencillo. Esta sección debe describir el principio fundamental que subyace al patrón. La solución está a su vez formada por las siguientes subsecciones:

- *Estructura de la solución* – Define a los participantes y la relación entre ellos. Da una especificación detallada de los aspectos estructurales del patrón. Esta sección puede tener diagramas de objetos o de clases.
- *Dinámica de la solución* – Describe escenarios típicos del comportamiento del patrón en tiempo de ejecución. Esta sección puede tener diagramas de secuencia o de interacción.

Usos conocidos Ejemplos del uso de un patrón, tomados de sistemas existentes. Es común, aunque polémico también, dar al menos tres ejemplos, ya que algunos autores opinan que de no observarse al menos esta cantidad de ejemplos claros y bien presentados, probablemente no debería calificarse como patrón.

Consecuencias Los beneficios proporcionados por el patrón, y cualquier riesgo potencial (BUSF96, pp. 11, 19 - 21).

2.4.2. Clasificación de Patrones de Software

Diferentes autores proponen distintas formas de clasificar a los patrones de software; principalmente las dominantes son las usadas en (GAMM94) y en (BUSF96). Aquí se utilizará la segunda, la cual agrupa los patrones de software en *niveles* de escala o abstracción.

2.4.2.1. Nivel alto (patrones arquitectónicos).

Un *patrón arquitectónico* expresa una estructura organizacional fundamental o esquema para sistemas de software. Provee un conjunto de subsistemas predefinidos, especifica sus responsabilidades, e incluye reglas y guías para organizar las relaciones entre ellos.

Los patrones arquitectónicos son plantillas para arquitecturas concretas de software. Especifican las propiedades estructurales a lo ancho del sistema en una aplicación, y tiene un impacto en la estructura de sus subsistemas. La selección de un patrón arquitectónico es, en consecuencia, una decisión fundamental cuando se desarrolla un sistema de software (BUSF96, p. 12).

2.4.2.2. Nivel medio (patrones de diseño)

Un *patrón de diseño* provee un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre ellos. Describe estructuras comúnmente recurrentes de componentes comunicantes que resuelven un problema general de diseño en un contexto particular (BUSF96, p. 13).

Los patrones de diseño pueden clasificarse en tres subgrupos a partir de su propósito (GAMM94): De creación, de estructura y de comportamiento.También pueden clasificarse a partir de su ámbito, el cual especifica si el patrón se aplica a clases o a objetos. Bajo esta clasificación, se observa que la gran mayoría de los patrones opera sobre el ámbito de los objetos, mientras que son muy pocos los que operan sobre clases.

2.4.2.3. Nivel bajo (patrones de codificación, patrones de programación o simplemente *modismos*)

Un *modismo* es un patrón de bajo nivel específico a un lenguaje de programación. Un *modismo* describe como implementar aspectos particulares de componentes o las relaciones entre ellos usando características del lenguaje dado (BUSF96, p. 14).

2.4.3. Modismos

Un modismo puede definirse de varias maneras:

- Es un patrón utilizado como solución para problemas de programación *específicos de la instrumentación* en un lenguaje particular (BUSF96, p. 346).
- Es una "expresión" reutilizable de semántica de programación, en el mismo sentido en que una clase es una unidad reutilizable de diseño y código. Los modismos sencillos son convenciones notacionales que con frecuencia se vuelven importantes en el diseño de programas (COPJ94, Preface).
- Es un patrón de bajo nivel, específico a un lenguaje de programación. Un modismo describe cómo instrumentar aspectos particulares de componentes, o las relaciones entre ellos usando las características del lenguaje dado(BUSF96, pp. 14, 345).
- Es un *patrón de programación* cuya forma se describe en términos de construcciones de un lenguaje de programación (APPB00, Kinds of Design Patterns).
- Es una técnica de programación específica de un paradigma o de lenguajes que llenan detalles (internos o externos) del comportamiento o la estructura de un componente (APPB00, Kinds of Design Patterns).

Más aún, en general un modismo tiene al lenguaje como parte de su contexto (en la Forma Alejandriana). Algunos modismos son especializaciones de patrones de diseño (HENK00).

¿Qué es lo que proveen los modismos?

Cuando se aprende un nuevo lenguaje de programación, no basta con dominar su sintaxis; se debe conocer también los pequeños *trucos* y *reglas secretas* del lenguaje para poder producir código de alta calidad y con eficiencia. Es aquí en donde un modismo puede ayudar a resolver un problema recurrente con el lenguaje de programación que se use.

En teoría, un conjunto de modismos hace mucho más sencillo ser productivo en un nuevo lenguaje de programación, dado que los modismos pueden utilizarse para enseñar cómo usar las características de un lenguaje de programación de manera efectiva al resolver un problema particular.

Adicionalmente, los modismos pueden servir para formar un lenguaje común de comunicación entre los miembros de uno o varios grupos de programación. Un equipo de programadores experimentados que ya han trabajado juntos durante algún tiempo, normalmente comparten una experiencia común al pensar en términos de un conjunto de modismos. Ya que cada modismo tiene un nombre, generalmente se le menciona de manera explícita cuando se usa o se hace referencia a él. Esto, además, tiende a simplificar la incorporación de nuevos miembros al equipo de programadores.

Portabilidad de modismos

La "portabilidad de modismos" se refiere a la capacidad de utilizar un modismo en diferentes lenguajes de programación. Algunos modismos se puede observar en familias de lenguajes que comparten alguna característica. Por ejemplo, C, C++ y C# (entre otros) comparten la característica de que sus arreglos tienen al cero como su primer índice; ésto hace que el modismo Fronteras Asimétricas (HENK98) sea válido en todos ellos.

Algunos modismos, como aquéllos que sirven para asignar nombres, pueden transferirse fácilmente entre lenguajes. Otros dependen de características del modelo en que se basa el lenguaje, y simplemente son inaplicables cuando se portan o traducen, lo cual se observa en lenguajes con tipificación fuerte y estática (v.g. C++ y Java), en contraste con los lenguajes con tipificación dinámica y débil (v.g Smalltalk y Lisp). Por ejemplo, el modismo *Asidera/Cuerpo Contado y Desapegado* (*deattached counted handle/body*), propuesto en (COPJ94), describe un mecanismo de conteo de referencias para C++, el cual es innecesario en lenguajes como Smalltalk y Java debido a la presencia de un recogedor de basura automático (*garbage collector*).

Algunas veces se requiere portar un modismo desde un lenguaje hacia otro distinto. La importación de modismos puede ofrecer un poder expresivo más grande al ofrecer soluciones en un lenguaje que explota características similares a las del lenguaje de origen, pero la cuales no habrían sido de otra manera consideradas parte del estilo en el lenguaje destino.

De cualquier manera, es importante comprender que esto no es sino una generalización, y que debe considerarse cuidadosamente la conveniencia y desventajas de portar un modismo entre lenguajes. Por ejemplo, la gran mayoría de las bibliotecas de C++ han sufrido de la aplicación inapropiada de modismos de Smalltalk, y lo mismo puede

decirse de muchos sistemas Java con respecto a C++(HENK00, p. 5).

Ejemplo de un modismo: Fronteras Asimétricas

A continuación, se presenta el modismo Fronteras Asimétricas (*Asimetric Boundaries*), propuesto originalmente en (HENK98, p. 26) para el lenguaje de programación C. Aquí se muestra la forma en que los modismos se presentan como patrones, utilizando una descripción basada en la forma POSA.

Nombre *Fronteras asimétricas.*

Modismos relacionados Fronteras simétricas.

Contexto En el lenguaje de programación C, se tiene un arreglo indiceado de manera directa por números enteros con primer elemento cero.

Problema ¿Cómo recorrer todos los elementos del arreglo exactamente una vez?

Solución Utilice un intervalo medio abierto que incluya al primer elemento y excluya al elemento cuyo índice es igual a la cantidad de elementos en el arreglo.

```
#define TAM 10

char arreglo[TAM];
int indice;

int main()
{
  // "blanqueado" usando fronteras asimetricas:
  for (indice=0; indice<TAM; indice++)
    {
      arreglo[indice]=0;
    }
  // ...
}
```

Este tipo de ciclo está claramente asociado con el tipo de estructura de datos que soporta (un arraglo). Mientras siempre se comience con el primer índice y la condición sea mantenerse estrictamente menor al número de elementos en el conjunto, no se producirá ningún error en tiempo de ejecución.

Consecuencias Al mantener el uso consistente de este patrón en todo el programa se disminuye el trabajo asociado a encontrar errores al recorrer conjuntos.

Otros ejemplos de modismos son: *Handle/Body* de C++ (COPJ98a), ComplexInterfacesNeedCloneable de Java (BROW00) y la *Orthodox Canonical Class Form* (Forma Canónica Ortodoxa para las Clases, o simplemente Forma Canónica Ortodoxa o FCO), a la cual se describe con mayor detalle en el siguiente capítulo.

2.5. Resumen

Las dificultades intrínsecas del desarrollo de software provocan que los proyectos de software no se finalicen (o al menos que no se terminen a tiempo), cuesten más de lo planeado originalmente, o que el producto final no tenga la calidad esperada. Como consecuencia, el software producido con un gran esfuerzo no se utiliza, se subutiliza, o no representa una solución viable al problema que se intentaba resolver. La solución más reciente y prometedora que se ha propuesto y utilizado hasta el momento son los Patrones de Software, inicialmente basados en la OO, y que a su vez aprovecha los lenguajes de programación para la expresión de métodos en las clases.

Dentro de los Patrones de Software, los modismos se presentan como un grupo particular de patrones para la programación que dependen de los lenguajes de programación(COPJ94).

En el siguiente capítulo se destaca la FCO, originalmente propuesta para C++ como un modismo en particular que sirve como base para la implementación de otros modismos en C++.

3. Trabajo relacionado

Las cosas sencillas deben ser simples y las cosas complejas deben ser posibles.
– Alan Kay

En este capítulo se describe el aspecto y la intención de los modismos Forma Canónica Ortodoxa para las clases de C++ (u *Orthodox Canonical Class Form*) y Objeto Canónico de Java. El objetivo es presentar estas expresiones como trabajo relacionado.

3.1. La Forma Canónica Ortodoxa en C++

Primero se define la FCO para posteriormente expresarla como un patrón de software. A continuación se describen los elementos que la componen para finalmente describir el tiempo de vida de las instancias de las clases que se adieren a la FCO.

3.1.1. Definición

La FCO es un modismo o patrón de programación originalmente propuesto para el lenguaje orientado a objetos C++ por Coplien en (COPJ94). Su principal utilidad es controlar y predecir en tiempo de programación el comportamiento en tiempo de ejecución para la creación, asignación y destrucción de objetos de una clase específica en C++.

La FCO se considera uno de los modismos más importantes de C++. Puede verse como una "receta" para formular clases; el resultado de seguirla es que las instancias (llámense objetos o variables) creadas a partir de las clases que la implementan puedan asignarse, declararse y pasarse como argumentos a otras funciones de la misma manera en que lo hace cualquier variable del lenguaje C. De hecho, este modismo establece un conjunto de lineamientos e indicaciones que, cuando son seguidos apropiadamente, producen clases cuyos objetos se comportan de manera muy similar a como lo hacen los tipos de datos interconstruidos por el compilador. La FCO tiene las ventajas de dar flexibilidad al uso de las clases en C++ y ayudar al programador a prevenir o eliminar el comportamiento no previsto de las instancias de las clases (COPJ94, p. 28).

El nombre de la FCO viene de la combinación de los siguientes conceptos: la palabra "canónica"que para este propósito significa "que proporciona un conjunto de reglas", en las cuales el compilador se apoya para generar código, y la palabra "ortodoxa" que para este propósito significa que "es la forma más directamente comprendida y apoyada" por

el lenguaje mismo. Así, la FCO se refiere a la manera de escribir clases en C++ de modo que se siga un conjunto de reglas determinadas, ya que de esa forma las clases de C++ son apoyadas de manera más directa por el compilador.

Normalmente la FCO en C++ se especifica mediante cuatro métodos especiales: constructor por omisión, constructor de copias, la sobrecarga del operador asignación y el destructor. Estos métodos determinan la forma de crear, asignar y destruir objetos de las clases en C++.

3.1.2. La expresión de la FCO en C++ como patrón de software

Nombre Forma Canónica Ortodoxa para las Clases de C++.
También conocido como Forma Canónica Ortodoxa (o simplemente FCO).

Modismos relacionados Objeto Canónico (*Canonical Object*) de Java.

Contexto La FCO *debe usarse* durante la creación de cualquier clase no trivial de un programa en C++, con el propósito de manejar la creciente complejidad de cada clase y de toda la jerarquía de clases a lo largo de la evolución del programa. Algunos otros modismos se construyen sobre la FCO, y estos deben usarse cuando la aplicación lo sugiera (COPJ94, p. 44).

Problema Frecuentemente, los métodos que el compilador provee por omisión provocan que los objetos se comporten de manera impredecible o inconsistente. El constructor por omisión provee de objetos cuyas variables no están inicializadas, y el compilador de C++ no verifica si esta inicialización se ha llevado acabo. El constructor de copias y la asignación provistos por el compilador realizan una *copia superficial*. La falta de un destructor adecuado puede provocar la pérdida de referencias (*dangling references*) impidiendo el posterior acceso a los recursos utilizados por algunos objetos.

Solución Proveer explícitamente una versión del constructor por omisión, constructor de copias, operador asignación y destructor por parte del programador, que tengan un comportamiento predecible y consistente de las instancias de la clase.

- *Estructura de la solución* – Se compone de:
  - Constructor por omisión – Define la creación de objetos de una clase, cuando se construyen arreglos de objetos. Se invoca con la instrucción **new** sin dar argumentos al tipo de objeto.
  - Constructor de copias – Usado cuando se desea crear un objeto similar a otro, el cual es pasado como argumento a **new**. Este constructor también es invocado de manera implícita por el compilador cuando un objeto es pasado como argumento *por valor* a una función.
  - Operador asignación – Usado cuando se desea que un objeto ya existente adquiera el estado (i.e. los datos) de otro.

- Destructor – Usado cuando el objeto no se necesita más durante la ejecución del programa y se desea liberar los recursos utilizados por éste para que estén disponibles en caso de ser requeridos.

- *Dinámica de la solución* – En general, y para nuestro propósito, el tiempo de vida de los objetos de C++ es como sigue:

  Un objeto comienza su existencia una vez que se termina la ejecución de su constructor (ya sea el constructor por omisión, el de copias, u otro) y termina una vez que su destructor ha terminado de ejecutarse. Durante su tiempo de vida útil, es decir, el periodo entre el fin de la ejecución del constructor y el principio de la ejecución del destructor, el objeto puede ser copiado o asignado un número no determinado de veces.

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* El propósito de este programa es mostrar la estructura y la
   dinámica de la FCO en C++ */

/* Este objeto asigna un valor aleatorio a su único atributo en el
   momento en que se crea, o copia el valor del atributo de otro
   objeto si es que se utiliza el constructor de copias en vez del
   constructor por omisión */
class obj {
public:
  // Método público para consultar el estado del objeto.
  long int getState() {
    return state;
  }

  // Constructor por omisión
  obj() {
    state = random();
    printf("    Construyendo al objecto almacenado en %d con el estado %d.\n",
   this, state);
  }

  // Constructor de copias
  obj(const obj &o) {
    state = o.state;
    printf("    Construyendo copia en %d que también tiene el estado %d.\n",
```

```
   this, state);
  }

  // Sobrecarga del operador asignación
  obj &operator=(const obj &o) {
    // ¿Por qué asignar algún objeto a si mismo?
    if (this == &o) {
      printf("    Advertencia: La asignación reflexiva no tiene propósito.\n");
    } else {
      this->state = o.state;
      printf("    El objeto asignado adquirió el estado %d.\n", state);
    }
    return *this;
  }

  // Destructor
  ~obj() {
    printf("    Destruyendo al objeto almacenado en %d con el estado %d.\n",
   this, state);
  }

private:
  long int state; // Único atributo de esta clase.
};


// Esta funcion solo sirve para pasarle objetos por referencia.
void byReference(obj *objeto){
  /* No hay copia en este caso */
}

// Esta funcion solo sirve para pasarle objetos por valor.
void byValue(obj o){
  /* Al pasar a 'o' por valor se crea una copia de 'o' de manera
  automática.  El constructor de copias es responsable de crear la
  copia de 'o'.  Al terminar la función serán invocado el destructor
  de 'o', porque este sale de ámbito. */
}

int main (){
  // Se desea hacer "reproducible" la ejecucción de este programa.
  srand(time(0));

  printf("\nEl caso más sencillo posible es el siguiente:\n");
```

```
{
  // El constructor se invoca al declarar
  obj simplestLife;
  // Al salir del bloque el objeto es destruido automáticamente.
}
printf("\n");

/* El constructor por omisión del objeto es invocado cuando se le
   declara. El objeto se almacena en la pila (no en el montón). Una
   vez que el objeto sale de ámbito, su memoria es reclamada y su
   destructor es invocado de manera automática, y sin intervención
   del programador. */

printf("\nGeneralmente el caso más sencillo es el siguiente:\n");
obj *simpleLifeObject; // simpleLifeObject es una _referencia_ al objeto.
simpleLifeObject = new obj;
// El objeto solamente se crea y se destruye inmediatamente después.
delete simpleLifeObject;
printf("\n");

printf("\nLigeramente menos sencillo; el objeto es copiado:\n");
obj *originalObject, *xerox;
originalObject = new obj;
xerox = new obj(*originalObject); // Se usa el constructor de copias.
delete originalObject;
delete xerox;

printf("\nSimilar al anterior; ");
printf("el objeto es copiado implicitamente al pasarlo por valor:\n");
obj *otherObject;
otherObject = new obj;
byValue(*otherObject); // El objeto referenciado es pasado por valor.
delete otherObject;
printf("\n");

printf("\nEn contraste, ");
printf("el objeto no es copiado cuando se pasa por referencia:\n");
obj *anOtherObject;
anOtherObject = new obj;
byReference(otherObject); // La función recibe una referencia al objeto.
delete otherObject;
printf("\n");

printf("\nLa asignación regresa un objeto:\n");
```

```
    obj LHSObject, RHSObject;
    LHSObject = RHSObject;
    // Estos dos objetos no se destruirán hasta que termine el programa.
    printf("\n");

    printf("\nFin del ejemplo.\n");
}
```

Consecuencias Las clases construidas de esta forma proporcionan a sus instancias la característica de ser más fáciles de controlar que aquéllas donde el programador no proporciona explícitamente a los elementos que componen a este modismo. Las clases que usan la FCO son más robustas y flexibles que aquéllas que no lo hacen. El inconveniente de la FCO es que algunas clases muy sencillas no necesitan todos los elementos definidos en este modismo.

3.1.3. Elementos (componentes o participantes) de la FCO en C++

A continuación se analiza cada uno de los elementos que conforman la FCO:

Constructor por omisión Este es el responsable de inicializar objetos cuando no se pasan parámetros al constructor. Como consecuencia de lo anterior, para arreglos de objetos, el constructor por omisión es responsable de inicializar los miembros del propio arreglo de objetos. En la subsección 2.3.1.4 en la página 15 se aclararan los beneficios de usar constructores y en este caso también se aplica lo ahí mencionado.

Constructor de copias La principal función de este constructor es generar el copiado de los argumentos pasados por valor a una función, así como regresar el resultado de una función o método cuando éste sea una instancia de la clase; lo cual sucede de manera transparente para el programador. Adicionalmente, cuando se desea crear un objeto igual a otro, la manera más natural de hacerlo en C++ es usando el constructor de copias. El nombre de *Constructor de Copias* se debe a que inicializa a los objetos por medio de la copia del estado de otro objeto del mismo tipo(MURR93). Cuando el programador no provee un constructor de copias, el compilador provee uno que realiza copias superficiales, pero el programador puede proveer uno que adicionalmente cuente referencias o que realice copias profundas (en vez de superficiales) o cualquier otra cosa(STRB94).

Operador asignación: De manera similar a lo que sucede con el constructor de copias, cuando se usa este operador también se hace que un objeto tenga el mismo estado que otro objeto del mismo tipo. Adicionalmente a esta tarea, el operador asignación podría ser responsable de llevar un conteo de referencias, o de la recuperación de recursos utilizados por el objeto que se encuentre al lado izquierdo de la asignación. Nótese que es responsabilidad del programador preservar el significado intuitivo de la asignación, ya que nada en el lenguaje o el compilador obligan a hacer algo sensato(COPJ94).

Destructor: El destructor debe recuperar cualquier recurso adquirido por el objeto durante de la ejecución de sus constructores, o durante la ejecución de cualquiera de sus funciones miembro, una vez que éste ha dejado de ser útil para el programa. En la sección 2.3 sobre orientación a objetos se profundiza respecto los beneficios de usar destructores.

3.1.4. El tiempo de vida (o dinámica) de un objeto

El tiempo de vida de un objeto se relaciona con los constructores y destructores de la siguiente manera:

- El compilador debe comenzar por reservar espacio en memoria para almacenar el nuevo objeto, copiar algunas partes de éste, e invocar a alguno de sus constructores. Se dice que el objeto comienza a existir cuando se termina la ejecución del constructor invocado.
- Una vez que el objeto se ha inicializado, éste puede ser copiado (por ejemplo, cuando se pasa por valor), referenciado (por ejemplo, cuando se pasa por referencia), modificado (por ejemplo, cuando se incrementa alguno de sus datos), o consultado. El número de veces que estas acciones pueden suceder no está determinado.
- Una vez que el objeto deja de ser útil al programa y se desea recuperar los recursos utilizados por él para que puedan estar disponibles para otros usos. Para ello, los recursos deben liberarse. Esto sucede cuando el destructor del objeto ha terminado de ejecutarse.

Los constructores y destructores pueden ser invocados de manera implícita (v.g. cuando se declara un objeto, o cuando éste sale de ámbito) o de manera explícita (v.g. cuando se invoca a los operadores `new` y `delete`). Ambos estilos no deben mezclarse.

3.2. El modismo Objeto Canónico en Java

Primero se define el Objeto Canónico (OC) para posteriormente expresarlo como un patrón de software. A continuación se describen sus elementos componentes y finalmente se describe el tiempo de vida de las instancias que cumplen con este modismo.

3.2.1. Definición

El modismo OC de Java es un patrón de programación propuesto por Bill Venners en (VENB98), quien lo nombró así porque representa la forma más sencilla que un objeto en Java debe tener. La idea detrás de este modismo es sugerir una funcionalidad básica que se le da a cualquier objeto que se diseñe. La intención de este modismo no es forzar a los programadores a dotar a cada objeto con esta funcionalidad, sino simplemente proponer un conjunto de servicios que casi todo objeto debería implementar. En otras palabras, se

invita a los programadores a hacer canónico todo nuevo objeto a menos de que se tengan razones específicas para no hacerlo en algún caso dado.

Un elemento fundamental de la OO es que la instancia de una subclase puede ser tratada como si fuera una instancia de una superclase. Debido a que todas las clases de Java son subclases de `java.lang.Object`, se sigue que cualquier objeto puede ser tratado como una instancia de `Object`. Cuando se invoca cualquiera de los métodos declarados en la clase `java.lang.Object` en un objeto, se está tratando a ese objeto como una instancia de `Object`. Esos métodos, como `clone()`, `equals()`, `toString()`, `wait()`, `notify()`, `finalize()`, `getClass()` y `hashCode()`, proveen servicios básicos comúnmente utilizados en objetos Java de todo tipo. Invocando a esos métodos se podría estar tratando a objetos de diversas clases como instancias de la clase `Object`.

En su conjunto, todas las actividades que comúnmente se realizan en los objetos de Java constituyen un conjunto de servicios que, en la mayoría de los casos, debería implementarse como métodos dentro de todas las clases que se diseñen. Adicionalmente, se podría querer *serializar* (*serialize*) un tipo de objeto para darle "persistencia". Esto significa permitir que las instancias de la clase puedan ser recobradas después de que el programa que las crea o modifica termina de ejecutarse.

3.2.2. El modismo Objeto Canónico en Java y su expresión como patrón de software

Nombre Modismo Objeto Canónico.
También conocido simplemente como Objeto Canónico.

Modismos relacionados Forma Canónica Ortodoxa de las clases en C++.

Contexto Siempre que se diseñe una nueva clase en Java, a menos de que existan razones específicas para no utilizar este modismo.

Problema Frecuentemente los métodos que la clase `Object` o el compilador proveen por omisión provocan que los objetos se comporten de manera impredecible o inconsistente En vez de utilizar los métodos `equals()` y `clone()` heredados por omisión desde la clase `Object`. También es frecuente que se desee recobrar el estado de un objeto entre invocaciones de programas, lo cual es conocido como *persistencia*.

Solución El programador de la clase proporciona implementaciones de los métodos `equals()`, `clone()` y la interfaz `Cloneable`, además de decidir si el objeto implementará `Serializable`, las cuales sean predecibles y consistentes. Adicionalmente se implementa la interfaz `Serializable` (VENB98).

- *Estructura de la solución* – Se compone de
  - Implementar la interfaz `Cloneable` (a menos de que la superclase ya la implemente o el objeto sea inmutable).

- Reemplazar `clone()` si la clase incluye variables de instancia las cuales en algún momento durante la vida de sus instancias pudieran tener referencias a objetos mutables.
- Reemplazar `equals()` de modo que la comparación sea semántica.
- Implementar la interfaz `Serializable` (a menos de que una superclase ya lo implemente).

- *Dinámica de la solución* – Los objetos canónicos pueden ser clonados, serializados y comparados de manera semántica. A diferencia de lo que sucede con un objeto que hereda sus métodos directamente desde `Object`, los objetos canónicos pueden realizar copias no superficiales y pueden ser comparados de manera diferente a la que realiza el operador de comparación ("=="), además de que están equipados para ser persistentes.

Ejemplo:

```
// El archivo Trabajador.java fué escrito (originalmente en inglés)
// por Bill Venners como ejemplo del Objeto Canónico.
import java.io.Serializable;
import java.util.Vector;

public class Trabajador implements Cloneable, Serializable {

    private String nombre;
    private Vector listaDeTareas;

    public Trabajador(String nombre, Vector listaDeTareas) {
        if (nombre == null || listaDeTareas == null) {
            throw new IllegalArgumentException();
        }
        this.nombre = nombre;
        this.listaDeTareas = listaDeTareas;
    }

    public Trabajador(String nombre) {
        this(nombre, new Vector());
    }

    public void ponNombre(String nombre) {
        if (nombre == null) {
            throw new IllegalArgumentException();
        }
        this.nombre = nombre;
    }
```

```
    public void agregaAListaObject tarea) {
        listaDeTareas.addElement(tarea);
    }

    public Object clone() {

// Realiza el clon básico
        Trabajador elClon = null;
        try {
            elClon = (Trabajador) super.clone();
        }
        catch (CloneNotSupportedException e) {
            // Jamás debe suceder
            throw new InternalError(e.toString());
        }

// Miembros mutables de Clone
        elClon.listaDeTareas = (Vector) listaDeTareas.clone();
        return elClon;
    }

    public boolean equals(Object o) {

        if (o == null) {
            return false;
        }

        Trabajador w;
        try {
            w = (Trabajador) o;
        }
        catch (ClassCastException e) {
            return false;
        }

        if (nombre.equals(w.nombre) && listaDeTareas.equals(w.listaDeTareas)) {
            return true;
        }
        return false;
    }

    //...
}
```

Consecuencias: El beneficio de los objetos canónicos es que son más flexibles y robustos (mas sencillos de comprender, usar y cambiar) que sus similares no canónicos. Los objetos canónicos ayudan a hacer el código Java más flexible porque están listos para ser manipulados en las formas en que los objetos de cualquier tipo son comúnmente manipulados.

Usar este modismo tiene la desventaja de producir clases voluminosas y propensas a errores; no solo se debe ser cuidadoso de que el nuevo método `equals()` sea reflexivo, transitivo y consistente consigo mismo[1]: además debe ser consistente con el método `clone()`, de modo que `a.equals(a.clone())` siempre sea verdadero.

3.2.3. Elementos (componentes o participantes) del modismo Objeto Canónico

A continuación se presentan los diferentes elementos del modismo OC en Java:

Implementar la interfaz `Cloneable` La implementación de `Cloneable` permite que los objetos sean copiables. El propósito de poder copiar el estado de un objeto en otro (ya sea nuevo o preexistente) consiste en que ambos objetos puedan cambiar de manera independiente. Por ello, no tiene sentido implementar la interfaz `Cloneable` para una clase cuyos objetos sean inmutables, ya que por definición, tanto el objeto original, como el clon tendrían siempre el mismo estado.

Reemplazar `clone()` Si la clase que se está implementando tiene variables de instancia que en algún momento pueden hacer referencia a objetos mutables (cuyo estado puede cambiar), entonces debe reemplazarse el método `clone()`. Si no se hace de esta manera, el método heredado de la superclase (o de `Object`) podría no hacer el tipo de copia que el programador desea. Cuando el programador implementa él mismo el método `clone()`, se asegura de producir clases en las que tiene pleno control sobre el tipo de copia a realizar; ya sea una copia superficial, una copia profunda, una combinación de las anteriores o cualquier otro comportamiento.

Reemplazar `equals()` Al reemplazar `equals()` de modo que se realice una comparación semántica de objetos, en vez de simplemente heredar el comportamiento desde la clase `Object`, se evita que `equals()` haga lo mismo que el operador de comparación ('==='). Si no se reemplaza la implementación de `equals()` en la clase o en una de sus superclases (excepto `Object`), entonces `equals()` simplemente realiza una comparación de referencias.

Implementar la interfaz `Serializable` Esta implementación permite a las instancias de la clase tener persistencia y además hace que éste pueda implementar alguna

[1] En este caso *consistente consigo mismo* significa que múltiples invocaciones al mismo método usando los mismos valores deben regresar el mismo resultado. Diferentes resultados con los mismos valores indican que el método es inconsistente consigo mismo.

otra característica útil (un ejemplo de esto sería un *JavaBean*). Si la clase no hereda la implementación de la interfaz por parte de su superclase, entonces debe implementarse en ella misma.

3.2.4. Tiempo de vida (o dinámica) de un objeto

El tiempo de vida de un objeto canónico es el siguiente:

- Un objeto comienza a existir una vez que el responsable de crearlo (ya sea `clone()`, un constructor u otro) termina de ejecutarse, regresando una referencia al nuevo objeto.
- Una vez que el objeto es creado, éste puede ser copiado (usando `clone()`), comparado (por ejemplo, cuando se invoca `equals()`) y transferido (al implementar `Serializable`, el objeto se podría insertar en una base de datos). El número de veces que estas acciones pueden suceder no está determinado.

Puede ser interesante observar que una vez que `clone()` produce una copia de un objeto, el original y su copia pueden variar de manera independiente. De manera análoga, un objeto puede variar de manera independiente a una copia guardada por alguno de los mecanismos permitidos por la interfaz `Serializable`.

3.3. Resumen

Los modismos FCO de C++ y OC de Java proponen la implementación de interfaces y la instrumentación de métodos y operaciones como una forma de producir instancias que garanticen tener un comportamiento predecible y controlable desde el momento de su creación y hasta el final de su vida útil para el programa. En ambos patrones destaca la obtención de control al momento de copiar datos, de modo que se establezca claramente el tipo de copia que se realiza (superficial, profunda o una mezcla de las dos anteriores). Consideran el resultado de comparar objetos de la clase que se implementa, de modo que el significado de la comparación sea consistente con el significado del objeto, además de que la comparación sea también consistente con la operación de copia. Las clases producidas de esta manera resultan adecuadas para formar bibliotecas y módulos, ya que son robustas y flexibles bajo cualquier circunstancia.

4. Forma Ortodoxa para las Clases de C#

La única manera de aprender un nuevo lenguaje de programación es escribiendo programas en él.
-Dennis Ritchie

En este capítulo se describe la Forma Ortodoxa para las Clases de C# (FOCC#) como una construcción programática para el lenguaje C# similar a la FCO de C++ y al OC de Java.

4.1. Definición

La FOCC# es un modismo propuesto para el lenguaje C# cuya intención es proveer a sus clases de una estructura básica que asegure un comportamiento predecible para la creación, copia y destrucción de sus instancias. Tal estructura básica es equiparable a la FCO de C++ y al OC de Java.

El objetivo de este modismo es dar un esquema general a partir del cual las clases de C# puedan formarse. Al instanciar las clases de C# se desea producir objetos que se comporten de manera predecible. Siguiendo esta estructura básica para formular las clases se producen objetos cuyo comportamiento en tiempo de ejecución queda definido en tiempo de programación.

En su conjunto, todas las actividades comunes en los objetos de C# constituyen un conjunto de servicios que, en la mayoría de los casos, debería construirse dentro de todas las clases. La pregunta que intenta responder este modismo es: ¿qué se necesita (como mínimo) para hacer que cualquier objeto definido en C# maneje los servicios comunes a todo objeto?

Se sugiere adoptar este modismo para la práctica propia de la programación en C#, o si se desarrolla un conjunto de guías de estilo para un equipo u organización, se podría incluir a este modismo como parte de esas guías.

4.2. La expresión de la FOCC# como patrón de software

Nombre Forma Ortodoxa para las Clases de C#.

Patrones relacionados Forma Canónica Ortodoxa, Objeto Canónico, así como el Patrón Liberar y el Patrón *using*.

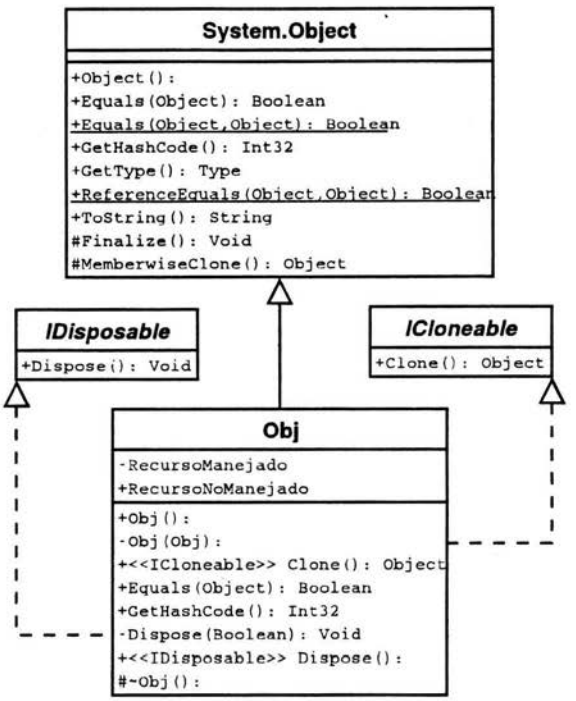
Contexto La FOCC# debe usarse durante la creación de cualquier clase no trivial de un programa en C#, a menos de que se tengan razones específicas para no hacerlo.

Problema Frecuentemente los métodos que el compilador o el ambiente de ejecución proveen por omisión provocan que los objetos se comporten de manera impredecible, inconsistente o ineficiente.

Solución El programador debe proveer implementaciones de ciertos métodos de importancia clave de modo que el comportamiento de los objetos sea predecible y consistente.

* *Estructura de la solución* – Cualquier clase no trivial requiere definir (o reemplazar la definición de) al menos a los siguientes miembros (organizados por grupos)
  - *Grupo de creación.*
    - Constructor por omisión.
  - *Grupo de copia y clonación.*
    - Construcción de copias y método `Clone()` para tipos referencia.
    - Implementación de la interfaz `IClonable` para producir copias profundas.
  - *Grupo de comparación.*
    - Reemplazar `Equals()`.
    - Reemplazar `GetHashCode()`.
  - *Grupo de destrucción.*
    - Implementar la interfaz `IDisposable`.
    - Instrumentar el destructor de la clase.

La siguiente figura muestra el diagrama de clases de la solución. La clase `Obj` es un ejemplo sencillo de una clase que utiliza la FOCC#. Al comparar las clases `Object` y `Obj` del diagrama se observa cuáles son los métodos de la clase `System.Object` que son reemplazados en la clase `Obj`. También se muestra la asociación entre interfaces, métodos y su visibilidad.



* *Dinámica de la solución* – Es posible clasificar la dinámica de la solución en dos categorías:

- *Dinámica externa*: Cuando un objeto que cumple con la FOCC# es creado, su constructor por omisión garantiza que se inicialice a un estado predecible y consistente. Posteriormente el objeto puede ser asignado o copiado produciendo siempre instancias consistentes en forma predecible. Los objetos ortodoxos (i.e. las instancias de clases que implementan la forma ortodoxa) pueden ser clonados con un tipo de copia consistente con la semántica del objeto y comparados de manera semántica usando `Equals()`, además que pueden usarse de otras formas comunes. Si se desea, el objeto puede ser guardado de manera persistente, de modo que el tiempo de vida del objeto puede trascender el tiempo de vida del proceso que lo creó originalmente. Finalmente, estos objetos están preparados para liberar sus recursos una vez que su existencia sea prescindible para el programa.
- *Dinámica interna*: El método destructor de una clase en C# produce un método protegido llamado `Finalize()` que reemplaza al método homónimo heredado desde la clase `System.Object`. Por otro lado, al implemen-

tar la interfaz `IDisposable`, se cuenta con el método público `Dispose()`, que al ser invocado sin parámetros invoca a su vez al método protegido `Dispose(Boolean)` de la misma clase. Asimismo, el constructor de copias (cuya visibilidad es privada) puede ser utilizado por el método público `Clone()` para producir copias de los objetos. De igual manera es frecuente que `Clone()` utilice internamente al método protegido llamado `MemberwiseClone()` que realiza copias superficiales. Finalmente, el método `Equals()` puede llamar a otros métodos `Equals()` sobrecargados en otras clases para determinar si el objeto actual (`this`) es igual a otro objeto del mismo tipo.

Ejemplo La clase que se muestra a continuación implementa los elementos de la FOCC# y es parte de un programa real (un programa que sirve para organizar las referencias bibliográficas de este mismo documento).

```
// Esta clase se adhiere a la FOCC# y es parte de un programa que
// verifica referencias bibliográficas en documentos de LyX/TeX/LaTeX.

using System;

// No contiene recursos no manejados, por lo cual no implementa
// IDisposable ni el destructor; sin embargo se adiere a la FOCC#.
class RefBib : ICloneable
{
  protected readonly string sId; // Por ser de solo lectura solo puede
 // modificarse desde un constructor.

  ////////////////////////////////////////////////////// Grupo de creación:
  // Constructor por omisión.
  RefBib()
  {
    // El constructor por omisión es privado, lo cual impide usarlo
    // desde fuera de la clase (disminuyendo errores).
  }

  // Constructor a partir de una cadena de caracteres.
  public RefBib(string s)
  {
    sId=s;
  }

  ////////////////////////////////////////////////// Grupo de copia y clonación:
  // Constructor de copias.
  RefBib(RefBib r)
```

```
  {
    this.sId=r.sId;
  }

  // Este método utiliza al constructor de copias.
  public Object Clone()
  {
    return new RefBib(this);
  }

  ////////////////////////////////////////////////////// Grupo de comparación:
  public override Boolean Equals(Object o)
  {
    // Si el objeto es nulo entonces es diferente al objeto actual:
    if(o == null) return false;
    //Si el tipo del objeto no es igual al de el objeto actual, son diferentes:
    if(this.GetType() != o.GetType()) return false;
    // Ahora se realiza una comparación miembro a miembro:
    if(this.sId != ((RefBib)o).sId) return false;
    // Los objetos son iguales:
    return true;
  }

  public override Int32 GetHashCode()
  {
    // El algoritmo utiliza al miembro sId, que es de solo lectura.
    Int32 hc=0;
    for(Int32 i=0; i<sId.Length; i++)
    {
      hc+=(sId[i].GetHashCode()*(i+1));
    }
    return hc;
  }

  // De manera similar a Equals() y GetHashCode(), ToString() también
  // puede ser ivocado sin intervención explícita del programador
  // (v.g. por WriteLine())
  public override string ToString()
  {
    return sId;
  }

}
```

Por razones de espacio no se muestra aquí el programa completo, pero este puede ser consultado en el apéndice A en la página 64.

Consecuencias

- Ventajas: Las clases construidas con estas características producen objetos mas fáciles de controlar que aquéllas que no implementan este modismo. Así, los objetos diseñados de esta manera se tornan flexibles (sencillos de comprender, usar y cambiar) y ayudan a hacer al código más fácil de modificar, ya que están listos para ser manipulados en las formas en que los objetos de cualquier tipo son comúnmente usados.
- Desventajas: Al tener tantos puntos a considerar, el programador puede equivocar la instrumentación si no es cuidadoso. Es especialmente frecuente escribir funciones de dispersión (`GetHashCode()`) que son lentas o equivocadas.

4.3. Participantes de la FOCC#

Los elementos que componen a la FOCC# se clasifican en cuatro grupos:

- Creación.
- Copia y clonación.
- Comparación.
- Destrucción.

Las categorías de copia y construcción tienen una aparente intersección en el constructor de copias, la cual se evita no declarando al constructor de copias como método público, mientras que las categorías de copia y comparación se relacionan porque las copias (producidas por el método `Clone()`) deben ser iguales al compararse (de acuerdo al método `Equals()`). Finalmente la destrucción de la clase puede realizarse de manera inversa a como se realiza la construcción de la misma, de modo que sea posible asegurar que una vez terminada la ejecución del destructor, cualquier recurso tomado por el objeto sea liberado nuevamente para su reutilización por otros objetos.

En las siguientes secciones se analizan los cuatro elementos que componen la FOCC# y se muestran ejemplos de cada uno de ellos. Es importante aclarar que los ejemplos no implementan a la FOCC# en su totalidad, sino solo a algunos de sus componentes. Los ejemplos no corresponden a código real correctamente escrito sino a ilustraciones de los elementos de la FOCC#. Tras analizar la participación de los elementos componentes y la dinámica con la cual interactúan, en el siguiente capítulo se muestra un ejemplo completo de una clase que de adhiere a la la FOCC#, donde se aprecia cómo se aplica cada uno de los elementos y también apreciar la dinámica entre ellos.

4.3.1. Constructor por omisión

Construir un objeto significa crearlo en memoria, es decir, reservar un espacio en memoria para contenerlo e inicializar tal espacio con los valores necesarios para que el nuevo objeto tenga un estado inicial predecible.

Para llevar a cabo la construcción de un objeto, su clase correspondiente debe definir un método *especial* llamado constructor. En C#, el constuctor es *especial* por tres razones:

- Un constructor no regresa valor alguno.
- En un constructor es posible modificar variables marcadas como *sólo para lectura* (`readonly`).
- Un constructor es invocado de manera automática para crear objetos usando la instrucción `new`.

Una clase en C# puede tener más de un constructor: todos los métodos constructores son homónimos a la clase y cada uno de ellos se distingue por el número y tipo de parámetros que recibe, con una excepción que se menciona a continuación:

4.3.1.1. Constructores sin parámetros

Una clase en C# puede instrumentar hasta dos constructores que no reciben parámetros: El *constructor de clase*, marcado como un constructor estático (`static`), y el *constructor por omisión*.

Cuando el programador define un constructor de clase, este es responsable de inicializarla. El constructor de clase, también conocido como constructor estático, siempre es privado (ya que no se puede invocar desde código escrito por el programador) y es único (debido que no recibe argumentos, no sería posible sobrecargarlo). En (RICJ02, pp. 187 - 190) de describe con mayor detalle al constructor de clase.

4.3.1.2. Constructor por omisión

El constructor por omisión es responsable de inicializar a las instancias de una clase cuando `new` se usa sin argumentos. Al proveer un constructor por omisión se garantiza que un objeto que no recibe argumentos en el momento de su creación no se encuentre en un estado impredecible o difícil de conocer.

Si el programador no define ningún constructor, entonces el compilador de C# provee un constructor por omisión vacío. El siguiente es un ejemplo de una clase que no instrumenta explícitamente un constructor por omisión:

```
class Obj
{
}
```

El código producido al compilar el ejemplo anterior es idéntico al producido al compilar el siguiente ejemplo[1]:

```
class Obj
{
  public Obj()
  {
  }
}
```

En casos sencillos (como los dos ejemplos anteriores) se observa que los constructores no son indispensables. Pero cuando se instrumentan clases que no son triviales, se hace evidente la conveniencia de instrumentar al constructor por omisión para propiciar que el programador indique explícitamente el estado que las instancias deben tener al momento de ser creadas. Esto le proporciona al programador control sobre sus clases, haciéndolas predecibles y controlables al momento de su instanciación. El siguiente es un ejemplo de una clase con un constructor que toma como argumento un `Int32` (nótese que este no es el constructor por omisión de la clase):

```
using System;
class Obj
{
  private Int32 i;
  public Obj(Int32 i)
  {
    this.i=i;
  }
}
```

Cuando el compilador de C# detecta que se instrumenta algún constructor, esto evita generar automáticamente al constructor por omisión para la clase. Como consecuencia el ejemplo anterior produce un error en tiempo de compilación si se intenta utilizar en un programa principal como el siguiente:

```
public class principal
{
  public static void Main()
  {
    Obj o = new Obj();
  }
}
```

[1] Lo cual puede corroborarse usando una herramienta como **monodis** o **ildasm**. Desensambladores de código intermedio como los anteriores son herramientas útiles para entender el comportamiento del compilador.

Para evitar este tipo de inconveniente, la FOCC# propone siempre implementar un constructor por omisión. El siguiente ejemplo modifica la clase anterior incluyendo un constructor por omisión:

```
using System;

class Obj
{
  private Int32 i;

  public Obj()
  {
  }

  public Obj(Int32 i)
  {
    this.i=i;
  }
}
```

4.3.2. Asignación, copia y clonación

Las operaciones de asignación, copia y clonación en C# están relacionadas entre sí. La asignación de tipos referencia es responsable de copiar referencias u objetos. La construcción de copias es una forma natural de C# en que se pueden crear nuevos objetos cuyo estado sea igual al de otros objetos de la misma clase, pero permite al programador modificar este comportamiento para producir resultados distintos. Finalmente, la clonación es responsable de crear un objeto idéntico a otro y su importancia radica en que algunas clases esperan interactuar con objetos de tipo `ICloneable`, de modo que invoquen a sus métodos `Clone()` sin intervención explicita por parte del programador.

En C#, cuando se asigna una instancia de un tipo referencia (las clases definidas por el programador son tipos referencia) no se crea un nuevo objeto; simplemente se crea una nueva referencia asociada al objeto original. En contraste, lo que sucede cuando se asigna un objeto de tipo valor (v.g. un `Int32` o una estructura como `DateTime`) es crear un nuevo objeto. Para que las instancias de las clases definidas por el programador realicen verdaderas copias, es conveniente implementar la interfaz `ICloneable`, es decir instrumentar un método llamado `Clone()` que regrese un nuevo objeto cuyo estado es igual al estado de `this`. Otras formas de copiar objetos son utilizar un constructor de copias o utilizar `MemberwiseClone()`.

El método `Clone()` frecuentemente se apoya en otros métodos para realizar sus copias. Cuando se desea producir copias superficiales generalmente se usa `System.Object.MemberwiseClone()`, pero para copias profundas se recomienda instrumentar un constructor de copias que no sea público, e invocarlo desde el interior de un método llamado `Clone()`.

4.3.2.1. Asignación

Asignar es asociar un valor a una variable. La manera de hacer esto en C# es utilizando el símbolo de asignación ("="). Un ejemplo muy sencillo es `a = 0`, donde `a` podría ser un objeto de tipo `Int32`; aquí se le esta asignando el valor cero. Un ejemplo ligeramente más complicado se observa cuando el valor de la derecha no corresponde a un tipo primitivo, y se requiere utilizar la instrucción **new** para indicar que el objeto que se asigna debe ser creado. Suponiendo que `o` sea una variable de tipo `Obj` definida por el programador, el ejemplo es así: `o = new Obj()`.

La asignación muestra un comportamiento distinto cuando los objetos involucrados son de tipo valor y cuando son de tipo referencia (ver p. 14). Mientras que la asignación de objetos de tipo valor produce una copia del objeto, la asignación de tipos referencia no crea una nueva copia de un objeto, sino una nueva referencia al objeto mismo.

Las clases definidas por el programador son de tipo referencia. El siguiente ejemplo muestra el comportamiento de la asignación de una clase definida por el programador:

```
using System;

class Obj
{
  public Int32 estado;

  Obj() // Las instancias de Obj se construyen con estado igual a cero.
  {
    estado = 0;
  }
}


class principal
{
  public static Main()
  {
    Obj o = new Obj();

    Obj o1 = o;
    Obj o2 = o;
    //              o.estado vale cero.
    o1.estado++; // o.estado vale uno.
    o2.estado++; // o.estado vale dos.
    // Las referencias o, o1 y o2 están asociadas a la misma instancia:
    System.Console.WriteLine(o.estado);
    // las referencias o1 y o2 no pueden variar independientemente.
  }
}
```

El inconveniente de la asignación para instancias de clases definidas por el programador es que las distintas referencias asociadas al mismo objeto no pueden variar de manera independiente entre ellas. En C# el comportamiento de la asignación no puede ser modificado por el programador para obtener en tipos referencia un comportamiento similar a la asignación de tipos valor (una manera de entender esto, es que no es posible sobrecargar el operador de asignación). Lo que el programador puede hacer es generar nuevas copias de los objetos, para posteriormente asignarlas utilizando la asignación usual de referencias en C#, mediante incluir un método de copia, la construcción de copias o la clonación.

Una manera de asignar dos objetos es definir un método que regrese un nuevo objeto con el mismo estado del objeto cuyo método es invocado. El siguiente es un ejemplo en el cual un método `Copia()` regresa un nuevo objeto. El nuevo objeto tiene el estado del objeto al cual pertenece el método invocado:

```
using System;

class Obj
{
  public Int32 estado = 0; // El estado es cero por omisión.

  // Regresa un nuevo objeto con el mismo estado.
  public Obj Copia()
  {
    Obj nuevo = new Obj();

    nuevo.estado = estado;
    return nuevo;
  }
}


class principal
{
  public static void Main()
  {
    Obj a = new Obj();
    Obj b;

    b=a.Copia();
    // ahora a y b no comparten su estado y pueden variar independientemente:
    a.estado--;
    b.estado++;
    Console.WriteLine(a.estado);
```

```
    Console.WriteLine(b.estado);
  }
}
```

Se observa en el ejemplo anterior que el nuevo objeto es creado con valores por omisión y que posteriormente se ajusta su estado para igualar al estado del objeto original. Sin embargo, este método tiene un inconveniente importante: Si la clase tiene algún miembro de solo lectura (**readonly**) la copia no podrá ser inicializada ya que solamente en los constructores es posible modificar a miembros de solo lectura.

Respecto al problema de la asignación, el método `Copia()` puede utilizarse para realizar una asignación profunda ya que entrega un objeto cuyo tipo es el mismo del objeto actual. De este modo, el método se aplica al objeto actual y la referencia resultante se asigna a la referencia de la copia.

4.3.2.2. Constructor de copias

Copiar un objeto es crear un nuevo objeto que tenga el mismo estado que otro objeto. Una solución más eficiente para la asignación de tipos referencia es usar un constructor que reciba como argumento a un objeto del mismo tipo del que se está creando (i.e. el tipo de la clase a la que pertenece). Este constructor es importante por ser útil para otros métodos que hacen copias y recibe el nombre especial de *constructor de copias*.

Se llama *constructor de copias* al constructor que espera como parámetro a un objeto del mismo tipo y cuyo propósito es igualar el estado del objeto actual (el que está siendo construido) con el estado del objeto recibido como parámetro. En el siguiente ejemplo se observa una implementación de un constructor de copias:

```
using System;

class Obj
{
  public Int32 estado;

  Obj() // Las instancias de Obj se construyen con estado igual a cero.
  {
    estado = 0;
  }

  // Construye el objeto actual de modo que el nuevo objeto tenga el mismo
  // estado que el objeto que recibio como parametro.
  Obj(Obj o)
  {
    estado = o.estado;
  }
```

```
}


class principal
{
  public static Main()
  {
    Obj o1 = new Obj();
    o1.estado++;
    Obj o2 = new Obj(o1);
    o2.estado++;

    // o2 se construyó con el estado de o1.
    System.Console.WriteLine(o2.estado);
    // Ombos objetos están separados y pueden variar independientemente.
  }
}
```

Se sugiere combinar las dos soluciones anteriores y hacer que el método `Copia()` utilice internamente al constructor de copias. Si adicionalmente se usa una interfaz como método de especificación, se consigue anunciar si la clase es copiable a través de un método estándar. En la siguiente subsección se analiza una solución como la aquí descrita.

Para resolver el problema de asignación de tipos referencia el constructor de copias puede utilizarse ya que permite la creación de un nuevo objeto con el estado de un objeto original pero asignado al nuevo objeto.

4.3.2.3. Clonación

Clonar un objeto es crear un nuevo objeto que tenga el mismo estado que otro objeto ya existente. Así, clonar un objeto es esencialmente copiar un objeto. Sin embargo hay diferencias entre la copia y la clonación. La importancia de la clonación en contraste con la copia de objetos es que las clases pueden anunciar que sus instancias son copiables de una manera estándar al implementar la interfaz `ICloneable`. En algún punto un programa puede preguntar si cierto objeto es copiable al verificar si implementa la interfaz `ICloneable` usando la instrucciones **as** o **is** de C#. Inclusive, ésta verificación podría realizarse sin intervención explícita del programador, ya que una biblioteca o módulo externo (no escrito por el mismo programador) podría realizar la verificación antes descrita.

Al escribir una clase es importante determinar si se desea permitir que más de una instancia de esa clase exista. De ser así, es conveniente que la clase implemente la interfaz `ICloneable`. El requisito para implementar la interfaz `IClonable` es instrumentar un método llamado `Clone()` que crea una copia del objeto que la invoca y regresa una referencia al nuevo objeto.

Copia superficial *versus* copia profunda[2]

El método privado `MemberwiseClone()` definido en la clase `System.Object` y heredado hacia cualquier objeto de C# realiza una copia superficial. A continuación se muestra un ejemplo de como implementar `Clone()` de modo que regrese una copia:

```
using System;

class Obj : ICloneable
{
  public Int32 estado = 0; // El estado es cero por omisión.

  // Regresa un nuevo objeto con el mismo estado.
  public Object Clone()
  {
    return MemberwiseClone();
  }
}


class principal
{
  public static void Main()
  {
    Obj a = new Obj();
    Obj b;

    b=(Obj)a.Clone(); // El "cast" se debe a que Clone() regresa un Object.
    // ahora a y b no comparten su estado y pueden variar independientemente:
    a.estado--;
    b.estado++;
    Console.WriteLine(a.estado);
    Console.WriteLine(b.estado);
  }
}
```

Se observa que este ejemplo es muy similar al analizado anteriormente para el método `Copia()`. De hecho, `MemberwiseClone()` tiene deficiencias similares; por ejemplo, cuando se requiere una copia profunda en vez de una superficial `MemberwiseClone()` no es suficiente, ya que los tipos referencia contenidos en el objeto copiado se copian superficialmente. Tampoco suele ser conveniente utilizar `MemberwiseClone()` en clases que usen

[2]En el apéndice en la página 78 se definen los términos *Copia Superficial* y *Copia Profunda*. Esas definiciones pueden complementar lo aquí expuesto.

eventos(NICK04). Otra deficiencia de **MemberwiseClone()** es que no puede escribir campos de solo lectura ya que esto solo se le permite a los constructores (como el constructor de copias que se analizó anteriormente).

Cuando se anuncia que la clase puede copiarse, y además, la copia que se requiere es profunda, no es suficiente simplemente invocar a **MemberwiseClone()**. Entonces es necesario instrumentar la copia recorriendo cada uno de los miembros de la clase, teniendo cuidado de hacer con cada uno de ellos el tipo de copia que se requiere. Este tipo de clonación puede apoyarse en **MemberwiseClone()**, en un constructor de copias, en **System.Reflection** o en una combinación de las anteriores para lograr un resultado que se ajuste por completo a la semántica de la clase.

El siguiente es un ejemplo de como instrumentar un método **Clone()** para hacer copias profundas:

```
class Objeto : ICloneable
{
    public int estado;
    Objeto()
    {
        estado = 0;
    }

    Objeto(Objeto o)
    {
        estado = o.estado
    }

    public Objeto Clone()
    {
        return new Objeto(this);
    }
}

Objeto o = new Objeto();
Objeto o1 = o.Clone();
Objeto o2 = o.Clone();
// Tanto o como o1 y o2 tienen estado igual a cero
o1.estado++; // o.estado y o2.estado valen cero mientras o1.estado vale uno.
o2.estado++; // o.estado vale cero mientras o1.estado y o2.estado valen uno.
// Las tres referencias están asociadas a diferentes instancias.
```

4.3.2.4. Resumen

Las clases que cumplan con la FOCC# no deben ignorar el tema de las copias. Ya sea que instrumenten un constructor de copias, que hagan privado ese método para no permitir las copias o que implementan la interfaz `IClonable` instrumentando `Clone()` con copias superficiales producidas por `MemberwiseClone()`.

El punto especialmente importante de la interfaz `ICloneable` es que al implementarla, la clase anuncia que es copiable, lo cual puede ser consultado por otras clases usando instrucciones como `as` e `is`. Un ejemplo podría ser una clase llamada `Bolsa` con un método `Guarda()` que solo acepte guardar objetos que implementen la interfaz `ICloneable`.

Finalmente se debe destacar que el programador es responsable de decidir una política de copiado; ya sea si realiza una copia superficial, una copia profunda, algún otro tipo de copia (v.g. parcialmente superficial y parcialmente profunda) o inclusive si se permite hacer copias en absoluto (e instrumentar los métodos que permitan llevar a cabo esta política de copiado). También debe destacarse que es responsabilidad del programador de la clase decidir si las instancias pueden copiarse o no.

4.3.3. Comparación y dispersión

Al comparar objetos, dos objetos son iguales si ambos tienen el mismo tipo y cada uno de sus atributos tienen, respectivamente, el mismo valor. Por lo anterior, cuando se define una clase, debe definirse un método `Equals()` para la misma. De no hacerlo la clase hereda la implementación por omisión desde su superclase, que en el caso del tipo `System.Object` significa que se realiza una comparación de referencias, es decir, se pregunta si los objetos son *idénticos*, es decir, que ambas se refieran al mismo objeto.

La comparación por igualdad de objetos es importante debido a que cualquier objeto puede usarse como llave en un `ArrayList` o en un `Hashtable`, el sistema en tiempo de ejecución del lenguaje mismo puede invocar a los métodos `Equals()` y `GetHashCode()` de los objetos llave sin intervención explícita por parte del programador; por ejemplo, al invocar a los métodos `Collections.ArrayList.Contains()` y `Collections.Hashtable.Add()`. Por ello, es importante que al definir un método `Equals()` también se defina el método `GetHashCode()` ya que de otra manera las instancias de la clase podrían comportarse de manera difícil de predecir. El código siguiente muestra como `Equals()` es invocado de manera implícita cuando se invoca el método `Collections.ArrayList.Contains()`.

```
using System;
using System.Collections;

class Obj
{
  public override Boolean Equals(Object o){
    Console.WriteLine("Equals invocado");
    return true; // o false; no importa cual.
  }
}
```

```
class principal
{
  public static void Main()
  {
    Obj a = new Obj();
    Obj b = new Obj();
    ArrayList al = new ArrayList();

    al.Add(a);
    al.Contains(b); // Esta linea provoca una comparación usando
    // Equals() sin que el programador la solicite de
    // manera explícita.
  }
}
```

Adicionalmente es común sobrecargar los operadores "==" y "!=" en términos de `Equals()` para que se comporten de manera acorde. Sin embargo los operadores "==" y "!=" no son invocados implícitamente por otras partes del sistema (contrastando con lo que sucede con `Equals()` y `GetHashCode()`) por lo cual sobrecargar los operadores "==" y "!=" se convierte simplemente en *azúcar sintáctico*, ya que no proporciona ventaja alguna sobre el uso de `Equals()` excepto que algunos programadores los consideran más fáciles de leer.

Al no estar forzosamente ligados el método y los operadores, es posible tener dos tipos de comparaciones: la proporcionada por el método y la proporcionada por los operadores. Si se toma la opción de mantener separados a los dos tipos de comparación, se debe ser muy cuidadoso ya que el programador podría confundir el tipo de comparación que requiere. Aunque ciertamente puede ser útil, por ejemplo, implementar una comparación superficial con el método y una comparación profunda con los operadores.

4.3.3.1. Igualdad

Al instrumentar el método `Equals()` debe cuidarse cumpla las siguientes reglas:

- Si `a` representa una instancia de una clase, entonces `a.Equals(a)` debe ser verdadero (reflexividad).
- Si `a` y `b` representan instancias de una clase y `a.Equals(b)` es verdadero, entonces `b.Equals(a)` debe ser verdadero (simetría).
- Si `a`, `b` y `c` representan instancias de una clase donde `a.Equals(b)` es verdadero y `b.Equals(c)` es verdadero, entonces `a.Equals(c)` debe ser verdadero (transitividad).
- La operación debe comportarse de manera consistente, es decir, si un par de objetos arrojan un valor dado (ya sea verdadero o falso) en un punto durante la ejecución

de un programa y en otro momento esos mismos objetos vuelven a ser comparados con el mismo método, entonces el valor de esa comparación no puede ser distinto al primero a menos de que el estado de alguno de los dos objetos involucrados no sea el mismo que cuando se realizó la primer comparación.

No cumplir con las reglas anteriores es arriesgarse a que las clases se comporten de manera impredecible; especialmente cuando las instancias de estos objetos quedan contenidas dentro de colecciones (v.g. listas, tablas, arreglos, diccionarios, etc.)

Igualdad de referencias (o identidad)

El tipo más sencillo de igualdad para tipos referencia es la identidad o igualdad de referencias. Para utilizar este tipo de comparación, el método `Equals()` puede invocar a su vez al método público y estático `ReferenceEquals()`, que se hereda desde `Object`.

Igualdad miembro a miembro

Dos objetos son iguales miembro a miembro si cada uno de los miembros de uno de los objetos es igual al respectivo miembro del otro objeto. El siguiente ejemplo muestra la comparación de dos objetos del mismo tipo que son iguales miembro a miembro:

```
using System;

// La clase base de la clase Punto es Object, pero la instrumentación de
// Equals() sería igual siempre que la clase base no reemplace al método
// Equals() heredado desde System.Object.
class Punto
{
  public Double primeraCoordenada = 0;
  public Double segundaCoordenada = 0;

  public override Boolean Equals(Object o)
  {
    // Si el objeto es nulo entonces es diferente al objeto actual:
    if(o == null) return false;
    //Si el tipo del objeto no es igual al de el objeto actual, son diferentes:
    if(o.GetType() != this.GetType()) return false;
    //Debido a la condición anterior, la siguiente conversión de tipo no falla:
    Punto p = (Punto) o;
    // Ahora se realiza una comparación miembro a miembro:
    if(primeraCoordenada != p.primeraCoordenada) return false;
    if(segundaCoordenada != p.segundaCoordenada) return false;
    // Los objetos son iguales:
    return true;
  }
```

```
}

class principal
{
  static Punto pS = new Punto();
  static Punto pT = new Punto();

  static Void compara()
  {
    Console.WriteLine("[{0},{1}].Equals([{2},{3}]) --> {4}",
      pS.primeraCoordenada,
      pS.segundaCoordenada,
      pT.primeraCoordenada,
      pT.segundaCoordenada,
      pS.Equals(pT));
  }

  public static Void Main()
  {
    // Con sus coordenadas inicializadas en cero, la comparación muestra que
    compara(); // ambos objetos verdaderamente son iguales (True).
    pT.segundaCoordenada=1; // Pero cuando se cambia una coordenada,
    compara(); // los objetos dejan de ser iguales (False).
  }
}
```

En algún caso podría desearse la comparación miembro a miembro entre objetos que no sean del mismo tipo, pero este tipo de comparación es la excepción, más que la regla. En el caso de la igualdad miembro a miembro es interesante observar la relación que este tipo de comparación guarda con `MemberwiseClone()`, ya que en una verdadera comparación miembro a miembro, si a representa un objeto y `b=a.MemberwiseClone()`, entonces `a.Equals(b)` siempre debe ser verdadero.

4.3.3.2. Dispersión – `GetHashCode()`

El método público `GetHashCode()` de la clase `System.Object` regresa un `Int32` usado por la clase `Collections.Hashtable` para insertar y recuperar elementos en tablas de dispersión. Por lo anterior, cuando se define una clase que instrumenta al método `Equals()` debe también redefinirse el método `GetHashCode()`. Al instrumentar el método `GetHashCode()` de una clase, se debe cuidar lo siguiente:

- Use un algoritmo que produzca valores esparcidos para lograr un buen desempeño de la tabla de dispersión [3].

[3]Las tablas de dispersión (*hash tables*) se explican de manera extensa en (KNUT98). De particular

- El algoritmo puede aprovechar el valor regresado por el método `GetHashCode()` de la clase base siempre y cuando la clase base no sea `System.Object`, ya que su implementación no es eficiente.
- El algoritmo debe utilizar, cuando menos, un miembro de instancia que sea de solo lectura (i.e. inicializado por el constructor). Frecuentemente esto es difícil, por lo cual debe hacerse un esfuerzo para no terminar con llaves de arreglos cuyos valores asociados no pueden ser consultados, porque la llave ya no se encuentra en la misma cubeta.
- El algoritmo debe ejecutarse de manera veloz. El uso intensivo de este algoritmo por arreglos hace que valga la pena invertir tiempo en el desarrollo de un método que termine de ejecutarse rápidamente.
- La función de dispersión debe ser inyectiva. En otras palabras: Si `a` y `b` son referencias a objetos, entonces si `a.Equals(b)` es verdadero, entonces `a.GetHashCode().Equals(b.GetHashCode())`debe ser verdadero.

Para una mejor comprensión sobre las funciones de dispersión se puede consultar la sección 3 (*Hashing*) del capítulo 6 (*Searching*) de(KNUT98).

Ejemplo

El código que aquí aparece muestra como `GetHashCode()` es invocado de manera implícita cuando se usa la tabla de dispersión `ht`. Tanto el método `Add()` como el método `Contains()` invocan a `GetHashCode()` una vez cada uno de ellos:

```
using System;
using System.Collections;

class Obj
{
  public override Int32 GetHashCode(){
    Console.WriteLine("GetHashCode() invocado");
    return 1; // una no muy buena función de dispersión.
  }
}

class principal
{
  public static void Main()
  {
    Obj a = new Obj();
```

interés para comprender la importancia del código de dispersión (*hash code*) es el uso de cubetas (*buckets*) para organizar el contenido de la tabla en grupos a partir de su código de dispersión.

```
    Hashtable ht = new Hashtable();
    // Las dos siguientes líneas invocan implícitamente a GetHashCode():
    ht.Add(a, "Contenido asociado a 'a'");
    ht.Contains(a);
  }
}
```

4.3.4. El destructor y la interfaz `IDisposable`

Destruir un objeto es recuperar sus recursos para permitir su reutilización por otros objetos. En C#, la recuperación de recursos manejados (es decir, memoria) está a cargo del recogedor de basura automático, dejando poco control del proceso al programador. La recuperación de recursos no manejados (es decir, descriptores de archivo, conexiones de red, etc., que generalmente son recursos escasos) queda a cargo del programador, quien debe encargarse de instrumentar lo necesario para que no se desperdicien tales recursos.

No todas las clases escritas por el programador necesitan tomar medidas especiales para recuperar recursos. Cuando una clase utiliza solamente recursos manejados por el recogedor de basura es mejor no tomar ninguna medida especial y permitir que el recogedor de basura tome el control. Cuando la clase utiliza recursos no manejados debe implementarse la interfaz `IDisposable` y el destructor, a riesgo de desperdiciar recursos de no hacerlo.

4.3.4.1. Liberación determinista de recursos

La liberación determinista de recursos es la capacidad de permitir que el programador decida en qué momento se libera el uso de un recurso escaso para permitir su reutilización por otra sección del programa. La liberación determinista contrasta con la liberación no determinista, que es usada por el recogedor de basura de C# para reutilizar la memoria.

El sistema en tiempo de ejecución de C# lleva un registro de los objetos cuya memoria puede ser liberada para su reutilización, pero no realiza esta liberación y reuso hasta que el sistema mismo lo determina. El programador no tiene control respecto al momento en que tal reutilización se realiza.

4.3.4.2. El problema de la liberación de recursos

En el caso de la memoria, no es necesario que el programador tenga control respecto al momento en que se realiza la reutilización de recursos, ya que el recogedor de basura se encarga por completo de ello. El caso de otros recursos, como los descriptores de archivos, es diferente: El recogedor de basura no cierra archivos cuando los descriptores de archivos comienzan a escasear. Así, es el programador el responsable de cerrar archivos una vez que no se necesitan más. Ya que el programador determina el momento exacto en que el archivo es cerrado, se dice que el descriptor de archivo asociado es liberado de manera determinista.

En consecuencia, cuando se programan clases que hacen uso de recursos no manejados por el recogedor de basura, es necesario que el programador de la clase incluya entre sus tareas la instrumentación de un método que permita la liberación determinista de los recursos no manejados de la clase. En otras palabras, si se programa una clase que abre un archivo (por ejemplo, durante la ejecución de su constructor) entonces debe instrumentarse un método (v.g. "`Cerrar()`", "`Liberar()`" o "`Dispose()`") que permita la liberación de sus recursos de manera determinista.

4.3.4.3. El patrón de diseño *Liberar*

La práctica estándar es instrumentar un método público que no recibe parámentros ni regresa valores llamado `Dispose()` (liberar). Si el nombre `Dispose()` no corresponde con la semántica del objeto, se recomienda adicionalmente escribir otro método también público y que tampoco reciba parámetros ni regrese valores cuyo nombre tenga sentido de acuerdo a la semántica de la clase. La instrumentación de este otro método simplemente invoca al método `Dispose()` y nada más.

El método `Dispose()` no debe producir excepciones ni errores en caso de ser invocado en más de una ocasión para cada instancia de la clase. Para lograr esto, el patrón de diseño Liberar(RICJ02; ARCT01) propone instrumentar `Dispose()` utilizando otro método homónimo, pero privado (en vez de público), y que espera un argumento booleano (en vez de ningún argumento).

Implementar correctamente este método no es sencillo, y frecuentemente las instrumentaciones del método `Dispose()` presentan errores. A continuación se muestra el patrón Liberar, que sirve como guía para instrumentar el método `Dispose()`

```
class obj
{
  static string s = "archivo.txt";
  FileStream fs;

  public obj(){
    fs = new FileStream(s,FileMode.Open);
  }


  public Void Dispose()
  {
    // Al liberar este objeto deja de ser necesario invocar a su finalizador.
    GC.SuppressFinalize(this);
    // Este método solo invoca al "verdadero" método de liberación.
    Dispose(true);
  }

  private void Dispose(Boolean desdeDispose){
```

```
    // Sincroniza hilos que lo invoquen simultáneamente
    lock (this) {
      if (desdeDispose) {
// Dispose fue invocado explícitamente.
// Desde esta sección del código es seguro acceder
// a miembros que hagan referencia a otros objetos
// porque el finalizador no se ha invocado.
      }
      // Hay que prevenir invocar a Close() sobre un objeto nulo.
      if(fs!=null) {
fs.Close(); // Dependiendo del tipo, en vez de Close()
//podría invocarse a Delete(), Dispose(), etc.
      }
      fs=null; // Podría no ser "null" sino otro valor "centinela".
    }
  }

}
```

4.3.4.4. Métodos `Finalize()` y destructor

`Finalize()` es un método (no público), declarado en la clase `System.Object` que se encarga de la "limpieza" final de un objeto en C#. La labor de "limpieza" se refiere a cerrar archivos o terminar conexiones de red que le objeto pudiera estar utilizando. Es especial dentro de C# porque el sistema en tiempo de ejecución garantiza que invoca al método `Finalize()` antes de que sus recursos sean reutilizados por otro objeto. A este método comúnmente se le llama *finalizador*.

Otro método especial es el llamado *destructor*. Este método no recibe parámetros ni regresa valor alguno. Su nombre es igual al de la clase, pero precedido por una tilde ("~") y se comporta de manera similar a un macro[4] o un *alias* para el finalizador. Cuando se define un destructor, el compilador genera el mismo código que hubiera generado si se definiera un método `Finalize()` con una estructura `try{...}finally{base.Finalize()}`, donde los puntos suspensivos indican el contenido completo del método destructor. El siguiente ejemplo utiliza un finalizador:

```
using System;
using System.IO;

class obj
{
  static string s = "/dev/null";
  FileStream fs;
```

[4]Una *macro* o *macrodefinición* es una definición global de renglones (LEVG01, p. 196). En otras palabras, una función que no recibe parámetros.

```
  public obj(){
    fs = new FileStream(s, FileMode.Open);
  }

  // Anula al finalizador heredado desde System.Object
  protected override void Finalize(){
    try {
      if(fs!=null) { fs.Close(); }
      Console.WriteLine("..fin...");
    }
    finally {
      base.Finalize();
    }
  }
}
```

El ejemplo anterior produce el mismo código que el siguiente ejemplo:

```
using System;
using System.IO;

class obj
{
  static string s = "ss-0.exe";
  FileStream fs;

    public obj(){
      FileStream fs = new FileStream(s,FileMode.Open);
    }

  // Destructor
  ~obj(){
    if(fs!=null) { fs.Close(); }
    Console.WriteLine("Destrucción o finalización.");
  }
}
```

Al compilarlos, los dos ejemplos anteriores producen exactamente el mismo código. Nótese que no es posible instrumentar un destructor y un finalizador en la misma clase; en realidad no todos los compiladores aceptan definir directamente el finalizador y exigen que se defina utilizando la sintaxis del destructor. Adicionalmente, C# no garantiza el orden en el que los finalizadores de diferentes objetos serán invocados, por lo cual los finalizadores no deben invocar métodos de otros objetos. Finalmente, nada impide escribir un mal método `Finalize()`, por lo cual se debe ser cuidadoso al escribirlo.

4.3.4.5. El patrón de diseño liberar y la interfaz `IDisposable`

Cuando se define una clase que implementa el método destructor o el método `Finalize()` se sugiere utilizar el *patrón de diseño Liberar* (*Dispose pattern*). Si una clase instrumenta este patrón y el método para liberar recursos se llama `Dispose()`, puede anunciar que libera recursos de manera determinista implementando la interfaz `IDisposable`.

La importancia de implementar la interfaz `IDisposable` radica en que algunas instrucciones de C# y clases de .NET hacen un uso especial de las clases que implementan la interfaz `IDisposable`. Así, el método `Dispose()` de una clase es invocado automáticamente cuando se aplica el modismo *using* (véase apéndice C en la página 76).

4.4. El tiempo de vida (dinámica) de un objeto en C#

El tiempo de vida de un objeto se relaciona con los constructores y destructores de la siguiente manera:

- Cuando un tipo de dato es creado, se le asigna espacio en la memoria disponible. La creación controlada de un objeto se realiza mediante implementar el constructor por omisión, por constructor de copias o por el método `Clone()`. En algún caso, el objeto podría "crearse" a partir de una representación persistente del mismo.

- Una vez que el objeto se ha inicializado, está listo para cambiar de estado. En otras palabras, es posible hacer referencia a éste, ya sea para consultarlo, meterlo en una colección (lo cual usa `GetHashCode()`), copiarlo (usando `Clone()` o el constructor de copias), compararlo (usando `Equals()` o el operador "==" sobrecargado), etc. Esto puede repetirse un número no determinado de veces.

- Finalmente, cuando un objeto ya no se necesita más, debe estar preparado para liberar sus recursos una vez que el recogedor de basura la solicite. Para esto, se utiliza el patrón liberar (que internamente usa los métodos `Dispose()`) en conjunción con el destructor (que internamente usa `Finalize()`).

Los constructores y destructores no siempre se invocaran de manera explícita; en algunas ocasiones estos pueden ser invocados de manera implícita al hacer una asignación al pasar un objeto como argumento a una función, etc.

4.5. Resumen

Las clases no triviales en C# deben tener ciertas características para garantizar que tienen un comportamiento predecible y controlable. Se considera que una clase en C# con tales características debe tener un constructor por omisión, debe implementar la interfaz `ICloneable`, debe invalidar las instrumentaciones de los métodos método `Equals()` y `GetHashCode()` heredadas desde la clase `System.Object`. Si la clase utiliza recursos no manejados debe implementar la interfaz `IDisposable` con el patrón de diseño liberar.

Las clases así construidas producen instancias que tienen un estado predecible y controlable desde el momento en que se crean, que pueden copiarse y que siempre se sabe que tipo de copias produce, ya sea superficiales, profundas o de otro tipo. Las instancias de estas clases siempre pueden compararse entre ellas de manera semántica, lo cual permite saber si los estados de dos instancias dadas son equivalentes. Finalmente las instancias de clases que cubren la FOCC# no desperdician recursos escasos y están preparadas para realizar una destrucción determinista.

Las clases ortodoxas canónicas no solamente se pueden usar en el programa para el cual se diseñaron originalmente, sino que estas clases pueden formar parte de bibliotecas o módulos y se comportan de manera confiable bajo cualquier circunstancia.

5. Conclusiones

Siempre que sea posible, robe código.
– Tom Duff

El presente trabajo desarrolla una expresión equivalente a la FCO de C++, pero traducida al lenguaje C#. El objetivo en todo momento ha sido conservar la mayor parte de la intención y los resultados de la versión original de Coplien para C++.

5.1. Resumen del trabajo

La Forma Ortodoxa para las Clases de C# (FOCC#) es un modismo propuesto para el lenguaje C# cuya intención es proveer a sus clases de una estructura básica que asegure un comportamiento predecible para la creación, copia y destrucción de sus instancias. Tal estructura básica es equiparable a la Forma Canónica Ortodoxa (FCO) de C++ y al Objeto Canónico (OC) de Java.

Así como los modismos FCO de C++ y OC de Java, la FOCC# propone la implementación de interfaces y la instrumentación de métodos y operaciones como una forma de producir instancias que garanticen tener un comportamiento predecible y controlable desde el momento de su creación y hasta el final de su vida útil para el programa. En este modismo destaca la obtención de control al momento de copiar datos, de modo que se establezca claramente el tipo de copia que se realiza (superficial, profunda o una mezcla de las dos anteriores). También considera el resultado de comparar objetos de la clase que se implementa, de modo que el significado de la comparación sea consistente con el significado del objeto, además de que la comparación sea también consistente con la operación de copia. Las clases en C# producidas de esta manera resultan adecuadas para formar bibliotecas y módulos, ya que son robustas y flexibles bajo cualquier circunstancia.

La FOCC# tiene los siguientes miembros organizados por grupos:

- *Grupo de creación.*
  - Constructor por omisión.
- *Grupo de copia y clonación.*
  - Construcción de copias y método `Clone()` para tipos referencia.
  - Implementación de la interfaz `IClonable` para producir copias profundas.

- *Grupo de comparación.*
  - Reemplazar `Equals()`.
  - Reemplazar `GetHashCode()`.
- *Grupo de destrucción.*
  - Implementar la interfaz `IDisposable`.
  - Instrumentar el método destructor.

Para el desarrollo de este modismo, la presente tesis se organiza como sigue:

- El capítulo 1 introduce al presente trabajo de tesis enunciando el problema a resolver, la hipótesis a considerar y las contribuciones de este trabajo.
- El capítulo 2 define propiamente la FCO tanto en su forma como en su función.
- El capítulo 3 analiza la expresión de la FCO en el lenguaje C++ y el trabajo relacionado con la FCO en el lenguaje Java.
- El capítulo 4 propone la Forma Ortodoxa para las Clases de C# (FOCC#)
- El capítulo 5 muestra un ejemplo que se desarrolla basado en los conceptos propuestos en esta tesis.

5.2. Revisión de la hipótesis

El propósito de este tesis ha sido responder a la siguiente pregunta como hipótesis:

¿Existe una expresión para la Forma Canónica Ortodoxa para las clases de C# que sea similar en forma (sintácticamente) y función (semánticamente) a la FCO propuesta originalmente por Coplien para las clases de C++?

La respuesta a la pregunta anterior es afirmativa y la forma se describe como la FOCC#.

5.3. Revisión de las contribuciones

La principal contribución de esta tesis es la descripción del modismo FOCC#. Este logra que las clases de C# produzcan objetos cuyo comportamiento sea predecible y controlable.

Adicionalmente, el presente trabajo contribuye a fomentar el uso de modismos en la práctica de la programación, como es el caso del patrón Liberar y la estructura `using(){}`.

5.4. Consideraciones finales

La conclusión a la que llega esta tesis es que en el lenguaje C# existe un modismo (al que se llamó Forma Ortodoxa para las Clases de C#) que transfiere a las clases que la implementan algunas de las mismas características que la Forma Canónica Ortodoxa para las Clases de C++.

Si bien ambos modismo solicitan que se instrumenten ciertos métodos, sólo los constructores son solicitado en ambas formas. Mientras que uno de los modismos solicita que se instrumente un destructor de instancias, el otro solicita que se implemente el patrón de diseño liberar. Por otro lado, la FOCC# pide que la comparación de objetos se realice de alguna manera que sea consistente con las copias de objetos producidas por el método `Clone()`.

La FOCC# toma elementos similares de la FCO de C++ (constructor por omisión, constructor de copias) y el OC de Java (`clone()`, `equals()`) agregando elementos particulares de C# (`IDisposable`, `GetHashCode()`). Se recomienda sugerir su uso en guías de estilo para equipos de programadores en C# dentro de la tecnología .NET.

A. Ejemplo

Cuídese de errores en el código anterior; solamente he demostrado que es correcto pero no lo he probado.
– Donald Knuth

```
// Esta clase se adhiere a la FOCC# y es parte de un programa que
// verifica referencias bibliográficas en documentos de LyX/TeX/LaTeX.

using System;

// No contiene recursos no manejados, por lo cual no implementa
// IDisposable ni el destructor; sin embargo se adiere a la FOCC#.
class RefBib : ICloneable
{
  protected readonly string sId; // Por ser de solo lectura solo puede
 // modificarse desde un constructor.

  ////////////////////////////////////////////////////// Grupo de creación:
  // Constructor por omisión.
  RefBib()
  {
    // El constructor por omisión es privado, lo cual impide usarlo
    // desde fuera de la clase (disminuyendo errores).
  }

  // Constructor a partir de una cadena de caracteres.
  public RefBib(string s)
  {
    sId=s;
  }

  ////////////////////////////////////////////////// Grupo de copia y clonación:
  // Constructor de copias.
  RefBib(RefBib r)
  {
```

```
    this.sId=r.sId;
  }

  // Este método utiliza al constructor de copias.
  public Object Clone()
  {
    return new RefBib(this);
  }

  ////////////////////////////////////////////////////// Grupo de comparación:
  public override Boolean Equals(Object o)
  {
    // Si el objeto es nulo entonces es diferente al objeto actual:
    if(o == null) return false;
    //Si el tipo del objeto no es igual al de el objeto actual, son diferentes:
    if(this.GetType() != o.GetType()) return false;
    // Ahora se realiza una comparación miembro a miembro:
    if(this.sId != ((RefBib)o).sId) return false;
    // Los objetos son iguales:
    return true;
  }

  public override Int32 GetHashCode()
  {
    // El algoritmo utiliza al miembro sId, que es de solo lectura.
    Int32 hc=0;
    for(Int32 i=0; i<sId.Length; i++)
    {
      hc+=(sId[i].GetHashCode()*(i+1));
    }
    return hc;
  }

  // De manera similar a Equals() y GetHashCode(), ToString() también
  // puede ser ivocado sin intervención explícita del programador
  // (v.g. por WriteLine())
  public override string ToString()
  {
    return sId;
  }

}
```

La sección anterior de código corresponde a la clase RefBib que se adhiere a la

A. *Ejemplo*

FOCC#. La siguiente sección corresponde a código que no se adhiere a la FOCC# y que es "cliente" o "usuario" de RefBib. Se desea hacer notar el manejo del archivo como un recurso no manejado para ilustrar como se podría hacer uso de un recurso no manejado en una clase que requiera hacer uso de la FOCC#.

```
// Este programa recibe como argumentos uno o más nombres de archivos
// .lyx o .tex y los revisa buscando citas bibliograficas y
// referencias bibliográficas. Al final muestra una lista con las
// refererencias hacia las cuales no se encontró alguna cita.

using System;
using System.IO;
using System.Collections;
using System.Text.RegularExpressions;

// Para compilar, este programa requiere asociarse a la clase "RefBib".

// Esta clase no se adhiere a la FOCC#: No vale la pena el esfuerzo de
// instrumentar destructor y finalizador si en este caso particular
// basta con utilizar la instrucción "using(){}" para controlar los
// recursos no manejados. Podría decirse que esta es una clase
// "trivial" y que no conviene hacer que cumpla con la FOCC#.
class Referencias
{
  static String sPatron  = @"([A-Z]{4}[0-9]{2})";
  static Regex expresionRegular = new Regex(sPatron);
  Hashtable ht = new Hashtable();

  // Este constructor va abriendo cada uno de los archivos de entrada
  // y revisa las referencias en su interior.
  public Referencias(String[] nombresArchivos)
  {
    foreach (String nombreArchivo in nombresArchivos) {
      using (StreamReader lector=new StreamReader(nombreArchivo)) {
revisaEntrada(lector);
      }
    }
  }

  Void revisaEntrada(StreamReader lector)
  {
    for(string linea=lector.ReadLine(); linea!=null; linea=lector.ReadLine())
    {
```

```
      // Compara el patrón de la expresión regular, contra la linea
      // recién leida desde el archivo de entrada.
      Match m = expresionRegular.Match(linea);
      while (m.Success)
      {
RefBib rb = new RefBib(m.ToString());
if(ht.Contains(rb))
{
  ht[rb] = (Int32)ht[rb]+1; // Incrementa el conteo de referencias.
}
else
{
  ht.Add(rb,1); // Agrega la primera de estas referencias.
}
m = m.NextMatch();
      }
    }
  }

  public Void muestraUnicas()
  {
    foreach (RefBib rb in ht.Keys)
    {
      if((Int32)ht[rb]<=2)
      {
Console.WriteLine("Referencia bibliografica sin utilizar: {0}.", rb);
      }
    }
  }

}

class principal
{
  public static Void Main(string[] args)
  {
    Referencias referencias=new Referencias(args);
    referencias.muestraUnicas();
  }
}
```

B. El patrón de diseño liberar y la interfaz IDisposable

La constante de un hombre es la variable de otro.
– Alan Perlish

En C# la recuperación de memoria está a cargo de el recogedor de basura automático, dejando poco control del proceso al programador. En contraste, la recuperación de recursos escasos no manejados queda a cargo del programador, quien debe encargarse de instrumentar lo necesario para que no se desperdicien los recursos no manejados por el recogedor de basura.

B.0.0.6. Que es la destrucción de objetos

Destruir un objeto es recuperar sus recursos para permitir su reutilización por otros objetos. El recogedor de basura es el responsable de recuperar los *recursos manejados* (que escencialmente es la memoria) ocupados por los objetos en desuso (v.g. objetos inalcanzables), pero no puede liberar el otro tipo de recursos, llamados *recursos no manejados* (v.g. descriptores de archivos), por lo cual las clases que usan (por uso), tienen (por agregación) o en son en si mismas (por herencia) [2.3.1.3 en la página 13] recursos distintos a la memoria deben instrumentar medidas especiales para prevenir el desperdicio de esos recursos.

B.0.0.7. Cuándo se destruyen los objetos en C#

Para objetos sencillos de C# (i.e., aquellos cuyo único recurso utilizado es memoria) el CLR (*Common Language Runtime* o sistema en tiempo de ejecución común a los lenguajes de .NET) se encarga de destruir al objeto liberando su memoria para que quede disponible para ser reutilizada por otros objetos. Se sabe que la destrucción del objeto sucede en algún momento posterior a que el CLR detecta que el objeto no se necesitará más, pero no es posible determinar exactamente cuando sucederá la destrucción; por lo anterior, se llama *no determinista* a este tipo de destrucción.

B.0.0.8. El problema con la destrucción no determinista

Cuando un objeto de C# que utiliza destrucción no determinista aloja algún recurso escaso no manejado se pueden producir problemas. En el siguiente ejemplo, el

programa eventualmente se queda sin descriptores de archivo y levanta la excepción ERROR_NO_MORE_FILES.

```
using System;
using System.IO;

class Objeto
{
  // El nombre de el ejecutable correspondiente a este archivo.
  static string s = "dest-det-0.exe";
  FileStream fs;

  public Objeto()
  {
    FileStream fs = new FileStream(s,FileMode.Open);
  }
}

// simplemente crea objetos.
public class DestDet
{
  public static void Main()
  {
    for(; true ;)
    {
      Objeto o = new Objeto();
    }
  }
}
```

El ejemplo anterior muestra la necesidad de algún tipo de destrucción determinista, que permita al programador controlar la liberación de recursos escasos, como los descriptores de archivos.

B.0.0.9. Destrucción determinista sencilla

Los objetos que ofrecen la capacidad de ser *cerrados*, *liberados* o *destruidos* de manera determinista implementan lo que se conoce como el *patrón de diseño Liberar* (*dispose pattern*). El patrón Liberar define convenciones a las que los programadores deben adherirse cuando definen un tipo que ofrecerá limpieza explícita a los clientes (o usuarios) de la clase. Adicionalmente, si un tipo implementa el patrón liberar, un programador que use el tipo sabe exactamente como limpiar explícitamente el objeto una vez que no se necesita mas(RICJ02, p. 472).

B.0.0.10. Destrucción determinista robusta

Las clases que implementan el patrón Liberar implementan un método publico que no regresa ningún valor y que no recibe argumentos y generalmente se llama `Dispose()`, `Close()` o `Clean()`. El nombre del método usualmente está asociado a la semántica del objeto y solo es convencional pero al llamar `Dispose()` al método se allana el camino para implementar la interfaz `IDisposable`, como se verá en la siguiente sección.

El método de destrucción `Dispose()` (o como sea que finalmente se llame) debe cumplir con ciertas características:

- Algún tipo de recurso puede usar una convención particular para denotar un estado que tiene alojado un recurso escaso y otro estado en el que no tiene el recurso; por ejemplo, un flujo de datos que pudiera encontrarse abierto o cerrado. Las clases que adopten esta convención pueden implementar un método público con un nombre adecuado (v.g. `Close()`) el cual a su vez invoca a un método con un nombre estándar como `Dispose()`.
- Durante su ejecución debe propagar la liberación de recursos por contención, de modo que si el objeto está compuesto de un archivo y un objeto que a su vez asigna recursos escasos, entonces los métodos `Close()` del archivo y `Dispose()` del objeto contenido deben ser invocados. Si la clase base implementa la interfaz `IDisposable`, entonces debe invocarse el métodos `Dispose()` de la clase base. Lo anterior se ilustra en el siguiente ejemplo:

```
using System;
using System.IO;

public class Contenido
{
  public void Dispose()
  {
    Console.WriteLine("El objeto contenido se está destruyendo ...");
  }
}

public class Contenedor
{

  FileStream fs = new FileStream("ejemploDestruccionConContencion.exe",
 FileMode.Open);
  Contenido c = new Contenido();

 public void Dispose()
 {
```

```
    Console.WriteLine("El objeto contenedor comienza su destrucción...");
    Console.WriteLine("Se comienza por cerrar el archivo...");
    fs.Close();
    Console.Write("Una vez cerrado el archivo, ");
    Console.WriteLine("se destruye el objeto contenido...");
    c.Dispose();
    Console.WriteLine("Ya no hay mas objetos contenidos sin destruir...");
    Console.WriteLine("El objeto contenedor termina su destrucción.");
    // Si la clase base de este objeto tuviera un método Dispose() o
    // si la clase base implementara la interfaz IDisposable(),
    // entonces al final debería invocarse a base.Dispose();.
  }
}

public class ejemploDestruccionConContencion
{
 static public void Main()
 {
    Contenedor o = new Contenedor();
    Console.Write("Se crea un objeto contenedor ");
    Console.WriteLine("y se destruye inmediatamente.");
    o.Dispose();
 }
}
```

- Si el método `Dispose()` de un objeto es invocado en más de una ocasión, el objeto debe ignorar cualquier invocación tras la primera. El objeto no debe lanzar ninguna excepción si su método `Dispose()` es invocado en múltiples ocasiones por uno o más hilos de ejecución. `Dispose()` puede lanzar una excepción si sucede un error debido a que algún recurso ya hubiera sido liberado sin que `Dispose()` hubiera sido invocado previamente. Lo anterior se ilustra en el siguiente ejemplo:

```
using System;
using System.IO;

public class Objeto
{

  FileStream fs = null; // Redundante ya que null es el valor por omisión.

  public Objeto()
  {
    fs = new FileStream("ejemploDisposeIdempotente.exe", FileMode.Open);
  }
```

```
  public void Dispose()
  {
    lock(this) { // Se necesita si la clase puede usarse con múltiples hilos.
      if(fs!=null) {
        Console.WriteLine("Se liberan los recursos no manejados.");
        fs.Close();
      } else {
        Console.WriteLine("Ya se habian liberado los recursos no manejados.");
      }
      fs=null; // null actua como un valor "centinela"
    }
  }
}

public class ejemploDisposeIdempotente
{
 static public void Main()
 {
   Objeto o = new Objeto();
   Console.WriteLine("Se crea un objeto y se destruye inmediatamente:");
   o.Dispose();
   Console.WriteLine("Si se intenta volver a destruir al mismo objeto:");
   o.Dispose();
 }
}
```

- Debido a que el método `Dispose()` debe llamarse explícitamente, los objetos que implementan `IDisposable` deben también instrumentar un finalizador para manejar la liberación de recursos cuando `Dispose()` no es invocado explícitamente por el programador. Por omisión, el recogedor de basura llama automáticamente al finalizador del objeto antes de recuperar su memoria. Una vez que el método `Dispose()` ha sido llamado, típicamente es innecesario que el recogedor de basura llame al finalizador del objeto que se ha liberado. Para prevenir la finalización automática, las instrumentaciones de `Dispose()` pueden llamar al método `GC.SupressFinalize()`. Lo anterior se ilustra en el siguiente ejemplo:

```
using System;
using System.IO;

public class Objeto
{
  protected void Dispose(Boolean desdeDispose)
  {
```

```
    if(desdeDispose) {
      // Aquí debe ir el código que realmente libera los recursos como se
      // mostró en los ejemplos anteriores.
      Console.WriteLine("Objeto destruido bajo el control del programador.");
    } else {
      // Este código es un "paracaidas" o una "bolsa de aire" en caso de que
      // suceda el accidente de no invocar al método Dispose() para terminar
      // la vida del objeto.
      // Debido a la naturaleza no-determinista del método Finalize(), no
      // debe hacerse referencia en esta sección a objetos cuyo método
      // Finalize() pudiera ya haber sido invocado.
      Console.WriteLine("Objeto destruido bajo el control del CLR.");
    }
  }

  public void Dispose()
  {
    GC.SuppressFinalize(this); // Ya no se necesita invocar el finalizador.
    Dispose(true);
  }

  protected void Finalize()
  {
    Dispose(false);
  }
}

public class ejemploDisposeFinalize
{
 static public void Main()
 {
   Objeto o = new Objeto();
   Console.WriteLine("Se crea un objeto y se destruye inmediatamente:");
   o.Dispose(); // Sin esta linea, el CLR invoca al método Finalize();
 }
}
```

B.0.0.11. Sintaxis del destructor

El último punto de la sección anterior ser refiere a la importancia de instrumentar el método `Finalize()` en las clases que implementan el patrón liberar. Para ello hay que tomar en cuenta que no todos los compiladores de C# permiten la llamar ni instrumentar "directamente" al método `Finalize()`. Para ello se utiliza un método especial llamado destructor.Cuando en C# se utiliza un destructor como en el siguiente ejemplo:

```
public class Objeto
{
  ~Objeto
  {
    ...
    // Código que invoca al método Dispose(false);
    ...
  }
}
```

Sintácticamente el destructor anterior significa lo siguiente:

```
public class Objeto
{
  protected Finalize()
  {
    try {
      ...
      // Código que invoca al método Dispose(false);
      ...
    }
    catch {
      base.Finalize();
    }
  }
}
```

Esta sintaxis de destrucción permite de manera sencilla manejar las excepciones en el código de limpieza sin olvidar invocar al finalizador de la clase base.

B.0.0.12. La interfaz IDisposable

Al implementar la interfaz `IDisposable` una clase anuncia que en su interior alberga recursos escasos y ofrece una manera determinista de recuperarlos. Las clases que implementan la interfaz `IDisposable` deben implementar un método público llamado `Dispose()` que no recibe argumentos y que no regresa ningún valor (i.e.: `public Void Dispose()`).

`Dispose()` debe usarse para cerrar, limpiar o liberar recursos no manejados (archivos, conexiones de red, conexiones de bases de datos, etc.) contenidos en un objeto que implementa la interfaz `IDisposable`. Por convención, este método es usado por todas las tareas asociadas con la liberación de recursos del objeto en preparación a su reuso.

B.0.0.13. El método Finalize()

El proceso que se sigue en preparación a la reutilización de la memoria es la finalización de un objeto. La finalización de un objeto se define en el método invalidable

`Finalize()` heredado desde la clase `System.Object` hacia todos los objetos. En el finalizador los objetos debe realizan lo necesario para liberar sus recursos no manejados y que así no se desperdicien.

Una alternativa al finalizador es el método destructor de la clase. De manera similar al constructor, el destructor es un método especial nombrado de la misma forma que la clase, pero precedido por el símbolo "~", que no recibe argumentos y tampoco regresa ningún valor. En C# el destructor de una clase funciona como una envoltura alrededor del método Finalize.

Cuando las clases definidas por el programador solo utilizan recursos manejados no deben implementar el método destructor para la clase ni el método Finalize() porque impone una carga extra al ambiente de ejecución. Cuando la clase utiliza recursos no gestionados (como conexiones a bases de datos, archivos, etc.) debe implementar un destructor porque eso permite liberar estos recursos escasos una vez que ya no se necesitan más.

La manera de lograr esto con C# es implementar la interfaz IDisposable. De esa manera, la clase estará obligada a implementar un método llamado Dispose el cual debe hacerse cargo de cerrar o desechar los recursos escasos que se encuentren en uso en el objeto.

C. El modismo *using*

Si hay duda, utilice fuerza bruta.
– Ken Thompson

Cuando se utiliza una clase que implementa la interfaz `IDisposable`, se recomienda invocar al método `Dispose()` en un bloque de manejo de excepciones `finally` para garantizar la liberación de sus recursos no manejados aun si durante el uso del objeto se produce una excepción. El correcto manejo de la invocación de `Dispose()` puede ser tedioso para el programador, así como una posible fuente de errores. La instrucción `using` de C# provee una manera mas conveniente de manejar esta situación.

A modo de ejemplo se muestra una clase que implementa la interfaz `IDisposable`. A continuación se muestra la forma en que se recomienda utilizar estas clases y finalmente se muestra una manera todavía más conveniente de dar el mismo tipo de manejo a las clases utilizando la instrucción `using`.

```
using System;

class obj : IDisposable
{
  public Void Dispose(){}
}
```

La clase anterior es el ejemplo más sencillo posible de una clase que implemente la interfaz `IDisposable`. Se recomienda que los clientes de la clases que implementen la interfaz `IDisposable` utilicen sus instancias dentro de un bloque `try` e invoquen a su método `Dispose()` dentro que un bloque `finally`. Como ejemplo se muestra en el siguiente código:

```
using System;

public class ejemploSinUsing
{
 static public void Main()
 {
   obj o = null;
   try {
     o = new obj();
```

```
    // En este punto el objeto "o" está correctamente creado (ya que
    // una excepción en su construcción lo habría sacado de este
    // bloque enviandolo al "finally") y se puede copiar, comparar,
    // etc.
  }
  finally {
    // Aun si se produce una excepción en el bloque "try",
    // o.Dispose() siempre es invocado.
    if (o != null) ((IDisposable) o).Dispose();
  }
 }
}
```

En comparación con el anterior, el siguiente ejemplo es más sencillo de escribir y utilizar; además es mas probable cometer errores al programar una estructura como la anterior, que al programar una estructura como la siguiente por lo cual se la prefiere:

```
public class ejemploConUsing
{
 static public void Main()
 {
  using (obj o = new obj()) {
    // En este punto el objeto "o" está correctamente creado y se puede
    // copiar, comparar, etc.
    // Al terminar este bloque, o.Dispose() es invocado automáticamente.
  }
 }
}
```

La utilización de la instrucción **using** produce código mas fácil de escribir, leer y corregir que el código que no la utiliza. Así, se recomienda que los clientes de clases que implementen la interfaz **IDisposable**, aprovechen el modismo *using*. Un corolario de lo anterior es: Los clientes de clases que se adieren a la FOCC# deben utilizar sus instancias dentro de bloques **using**.

D. Glosario

Nunca subestime el ancho de banda de una camioneta llena de cintas moviéndose a velocidad constante por la carretera.
– Andrew S. Tanenbaum.

.NET .NET es una iniciativa propuesta por Microsoft y estandarizada por la ECMA, dirigida a conectar dispositivos, información y personas. Esta iniciativa simplifica a los desarrolladores de software construir más que aplicaciones de Web; también facilita el desarrollo de aplicaciones gráficas, aplicaciones de consola, servicios Web, etc.

Alcance O visibilidad (*scope*). Se refiere a la región de un programa en la cual cierta variable es "visible", es decir, que su valor puede ser consultado.

Ámbito ALCANCE

C Lenguaje especializado en la programación de sistemas. Diseñado a principios de la década de 1970 por Dennis Ritchie, de los Laboratorios Bell, en New Jersey, Estados Unidos. Se emplea comúnmente para escribir compiladores y sistemas operativos(LEVG01, P. 247).

C++ Es una versión extendida del lenguaje C, diseñado hacia 1982 por Bjarne Stroustrup, de los Laboratorios Bell. Permite la programación orientada a objetos (OOP: *Object Oriented Programming*), además de emplear recursos gráficos estandarizados. Gran parte de los sistemas modernos de todo tipo están escritos en este lenguaje(LEVG01, P. 247).

C# Un lenguaje de programación orientado a objetos con un sistema seguro de tipos apoyado por Microsoft para ser usado con el *Framework* de .NET. C# (que en inglés se pronuncia como "si-*sharp*") fue creado específicamente para construir aplicaciones de escala empresarial usando el Framework de .NET. Es similar en sintaxis tanto a C++ como a Java y es considerado por Microsoft como la evolución natural de los lenguajes C y C++. C# fué creado por Anders Hejlsberg (autor de Turbo Pascal y arquitecto de Delphi), Scot Wiltamuth y Peter Golde. C# está definido por el estándar ECMA-334 (MURJ03).

Canónico, *ca* (adj.) Que se ajusta a las características de un canon de normalidad o perfección(ELMU03).

CIL Lenguaje Intermedio Común (*Common Intermediate Language*). Es un lenguaje que permite a los lenguajes de .NET total interacción entre sus objetos y el *framework*, además de gran independencia de la arquitectura (procesador) y la plataforma (sistema operativo) donde finalmente se ejecuta el código intermedio.

Clase Una pieza de construcción fundamental en los lenguajes orientados a objetos. Una clase especifica y encapsula sus estructuras de datos internas así como la funcionalidad de sus *ejemplares* u *objetos*. Una descripción de clase puede construir una o más clases por *herencia*.(BUSF96)

CLR Sistema en tiempo de ejecución común a los lenguajes de .NET (*Common Language Runtime*). Es el sistema que permite la ejecución de *código intermedio común* producido por los lenguejes de .NET.

Copia profunda Un objeto A es una *copia profunda* de otro objeto B cuando todos los datos contenidos en A (los cuales le proporcionan su estado), que llamaremos $A_1, ..., A_n$son independientes de los datos $B_1, ..., B_n$contenidos en el objeto B, de modo que ninguna modificación en algún miembro $A_m \in A$produce un cambio en el estado del objeto B. Las copias profundas son mas lentas de realizar que las copias superficiales y generalmente son mas difíciles de escribir, proporcionando un punto más donde el programador puede equivocarse. Por otro lado, las copias profundas tienen la característica de que una modificación al objeto original, no cambia el estado de la copia ni viceversa.

Copia superficial Un objeto A es una *copia superficial* de otro objeto B cuando cuando no todos los datos contenidos en A (los cuales le proporcionan su estado), que llamaremos $A_1, ..., A_n$son independientes de los datos $B_1, ..., B_n$contenidos en B. Las copias superficiales se realizan más rápidamente que las copias profundas y generalmente son más sencillas de programar, pero tienen el inconveniente de que modificar el estado del objeto A puede modificar el estado del objeto B y viceversa.

ECMA Asociación Europea de Fabricantes de Computadoras (*European Computer Manufacturer Association*).

Ejemplar Un *objeto* originado a partir de una clase específica. Frecuentemente usado como sinónimo de *objeto* en un ambiente orientado a objetos. Este término puede también ser utilizado en otros contextos (ver *ejemplificación*)(BUSF96).

Ejemplificación Un mecanismo que crea un nuevo *ejemplar* a partir de alguna plantilla. El término es usado en muchos contextos. Los *objetos* son ejemplificados a partir de *clases*. Las plantillas de C++ son ejemplificadas para crear nuevas clases o funciones. Un *marco de aplicación* es ejemplificado para crear una *aplicación*. La frase "ejemplificación de un patrón" es algunas veces usada para hacer referencia a tomar la descripción de un patrón, y rellenar los detalles necesarios para ajustarse a alguna aplicación específica(BUSF96).

Framework Es una colección organizada de objetos enfocados hacia un dominio específico que además provee interfaces apropiadas para la adaptabilidad hacia el interior de otras colecciones organizadas de objetos. Lo anterior es una simplificación del concepto de *framework* derivada de (ALHS98, p. 68). Para entender mejor el complejo concepto de *framework* se recomienda leer (CLIM03) ya que el autor enfoca su artículo hacia .NET argumentando que no es un *framework*.

Heap MONTÓN

Herencia Característica de los lenguajes orientados a objetos que permite que nuevas *clases* se deriven de las ya existentes. La herencia define el reuso de la implementación, una relación de subtipo, o ambos. Dependiendo del lenguaje de programación, la herencia es sencilla (cuando alguna clase hereda de una o cero clases) o múltiple (cuando una clase hereda de dos o más clases)(BUSF96).

Idiom MODISMO

Interfaces INTERFAZ

Interfaz El conjunto de propiedades y comportamientos que un objeto exhibe a su exterior. En C++ se consideran niveles en el control de acceso, con lo cual lo declarado como público (`public:`) es accesibles a cualquier objeto, lo declarado como protegido (`protected:`) es accesible solamente a otros objetos de la misma clase y lo declarado como privado (`private:`) es inaccesible a cualquier objeto. Por su parte C# agrega niveles al control de acceso de C++, y de manera más importante, agrega al lenguaje el concepto de interfaz, permitiendo que un objeto *implemente* una interfaz cuando la clase a la que pertenece cumple con tener públicamente disponibles un conjunto de métodos y propiedades especificados en una estructura `interface`.

Java Lenguaje de programación diseñado en la década de 1990 por James Gosling de Sun Microsystems, Java es un lenguaje orientado a objetos cuyas principales influencias se encuentran en C++ y Smalltalk. Opera con una gran independencia de la plataforma en la que se ejecuta, gracias a que usa una máquina virtual que interpreta código intermedio.

JavaBean Un tipo de componente de Java que cumple con algunas características dadas como, por ejemplo, la de implementar la interfaz `Serializable`. En particular, una clase que implementa la interfaz `Serializable` y el constructor sin argumentos es un JavaBean(VENB98).

Lenguaje Se llamará simplemente "Lenguaje" con mayúscula a un "lenguaje de programación de computadoras", como pueden ser C, C++, C# o Java.

Mensaje Los mensajes se usan para la comunicación entre *objetos* o procesos. En un sistema orientado a objetos el término mensaje se usa para describir la selección y activación de una operación o *método* de un objeto. Esta clase de mensaje es

síncrono, lo cual significa que el remitente espera hasta que el destinatario termina la operación activada.

Método Denota una operación realizada por un *objeto*. Un método se especifica dentro de una *clase*(BUSF96).

Modismo Del Inglés "Idiom". Los modismos son una tipo especial de patrones que están asociados a un lenguaje de programación específico.(COPJ98, 10:) La sección [2.4.2.3 en la página 19] contiene una extensa explicación.

Modismo A los patrones de bajo nivel, específicos de a algún lenguaje de programación se les llama modismos. Un modismo describe como implementar aspectos particulares de componentes o de las relaciones entre ellos con las características de un lenguaje de programación dado (BUSF96, P. 345).

Mono El proyecto Mono es una iniciativa de desarrollo abierto patrocinada por Ximian que se esfuerza por desarrollar una versión abierta de la plataforma de desarrollo .NET para Unix. Su objetivo es permitir a desarrolladores de Unix el construir y utilizar aplicaciones .NET entre distintas plataformas. Es posible encontrar más información al respecto en `http://www.go-mono.com`.

Montón Es la zona de memoria donde se alojan los objetos cuyo tiempo de vida no depende de su alcance.

Object OBJETO

Objeto Es la combinación de un estado y un conjunto de métodos que explícitamente conforman una abstracción que se caracteriza por su comportamiento ante ciertas solicitudes. Un objeto es un ejemplar de una clase. Un objeto modela una entidad del mundo real y se implementa como una entidad computacional que encapsula estado y operaciones (internamente implementados como datos y métodos) y responde a solicitudes de servicios(OMAG92).

Ortodoxo, xa (adj. y s.) Conforme con la doctrina tradicional en cualquier rama del saber(ELMU03).

Patterns PATRONES

Patrones En la sección 2.4 en la página 17 se describen y definen y clasifican los patrones de software.

Patrones de software PATRONES

Persistencia Es la capacidad de que un objeto exista más allá de la ejecución de un programa. La *serialización* es la clave para la implementación de la persistencia(JAWJ99, p. 811).

POO Programación orientada a objetos.

Serialize SERIALIZAR, SERIALIZACIÓN

Serialización La serialización de objetos permite escribir un objeto en un flujo así como leer un objeto desde un flujo. La serialización se usa comúnmente para almacenar objetos en archivos, pero también sirve para migrar objetos entre computadoras colaborando así en el balanceo de cargas de un sistema distribuido(JAWJ99, p. 811).

Shallow copy COPIA SUPERFICIAL

Signo Cualquiera de los caracteres que se emplean en la escritura y en la imprenta: signo matemático, musical; $ es el signo del dólar. Señal o figura que se usa en matemáticas para indicar la naturaleza de las cantidades o las operaciones que se han de ejecutar con ellas: el aspa es el signo de la multiplicación(ELMU03).

Scope ALCANCE

Visibilidad ALCANCE

OO Orientación a Objetos.

E. Referencias (Libros, revistas, etc)

Cualquier imprecisión en este índice puede explicarse por el hecho de que ha sido ordenado con la ayuda de una computadora.
– Donald Knuth

Bibliografía

[ALEC77] Alexander, Christopher (1977), A Pattern Language.

[ALHS98] Alhir, Sinan Si (1998), UML In A Nutshell. O'Reilly.

[APPB00] Appleton, Brad (14/Feb/2000). http://www.enteract.com/~bradapp/docs/patterns-intro.html

[ARCT01] Archer, Tom, A Fondo C#. McGraw-Hill, España 2001.

[BALR] Baldwin, Richard. http://www.developer.com/java/article.php/995731

[BOHB81] Bohem, Barry W. (1981). Software Engineering Economics. Prentice Hall.

[BROF75] Brooks, Frederick P. (1975). The Mythical Man-Month. Addison-Wesley, Reading, MA.

[BROW00] Brown, Kyle (16/Mar/2000). http://c2.com/cgi/wiki?ComplexInterfacesNeedCloneable

[BUSF96] Buschmann, Frank, et all (1996). Pattern-Oriented Software Architecture: A system of patterns Willey, England.

[CLIM03] Clifton, Marc (4/Nov/2003). http://www.codeproject.com/gen/design/WhatIsAFramework.asp

[COA92] Coad,O.: Object-Oriented Patterns, Communications of the ACM, Col. 35, No. 9, Sept. 1992, pp.152-159.

[COPJ94] Coplien, James (1994). Advanced C++: Programming styles and idioms, Addison-Wesley, EE. UU., Correcciones a la edición de 1992.

[COPJ98] Coplien, James. Software Design Patterns: Common Questions and Answers.

[COPJ98a] Coplien, James (1998) C++ Idioms. Lucent Technologies, Bell Laboratories.

[COPJ00] Coplien, James (2000) Foundation of Pattern Concepts and Pattern Writing. XV Simpósio Brasileiro de Engenharia de Software. Tutorial 1. Brazil 2001.

[COPJ01] Coplien, James. http://hillside.net/patterns/ (5/Oct/2001)

[ELMU03] El Mundo, Diccionario de la Lengua Española, http://www.elmundo.es/diccionarios/

[GABR01] Gabriel, Richard (5/Oct/2001) http://hillside.net/patterns/

[GAMM94] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994) Design Patterns: Elements of Reusable Object-Oriented Systems. Addison-Wesley. Reading, Massachusets.

[HENK98] Henney, Kevlin (1998). Idioms, breaking the language barrier.

[HENK00] Henney, Kevlin (Jul/2000). Substitutability: Principles, Idioms, and Techniques for C++. Dr. Dobb's Journal.

[HOFD79] Hofstadter, Douglas R. (1979). Godel, Escher, Bach : An Eternal Golden Braid. Vintage.

[JAWJ99] Jawroski, Jamie (1999). Java 1.2 Al descubierto. Prentice Hall. Madrid.

[KNUT98] Knuth, Donald (1998). The Art of Computer Programming. Volume 3: Sorting and Searching. Addison Wesley, EE. UU.

[PARD72] Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules.

[LEVG01] Levine, Guillermo (2001) Computación y programación moderna: Perspectiva integral de la inf Addison-Wesley, México.

[LITT01] Littlefair, Tim (2001). An investigation into the use of software code metrics in the industrial s Edith Cowan University. Australia.

[MURJ03] Murray, James D., Microsoft .NET Glossary., http://www.developer.com/net/cplus/article.php/1756291

[MURR93] Murray, Robert (1993). C++ Strategies and Tactics. Addison-Wesley.

[NICK04] Nicholas, Keith (Abr/2004) http://www.thecodeproject.com/csharp/EventsCopyFromsClones.a

[OMAG92] Object Management Group (Sept. 1992) Object Management Architecture Guide, Ed 2.0.

[ORTJ96] Ortega Arjona, Jorge Luis (1996). Estudio y evaluación de la programación orientada a objetos Tesis UNAM, México D. F.

[PRAP02] Parks, Paul. (16/Mar/2002) The Undernet C# channel FAQ - Classes and objects. http://www.parkscomputing.com/dotnet/csharpfaq/oo/

[PRYN] Pryce, Nat. http://www.doc.ic.ac.uk/~np2/patterns/tcl/index.html

[RICJ02] Richter, Jeffrey (2002) Applied Microsoft .NET Framework Programming. Microsoft Press, EE. UU.

[STRB91] Stroustrup, Bjarne (1991) The C++ Programming Language. (Second Edition). Addison-Wesley, EE.UU., 1995.

[STRB94] Stroustrup, Bjarne (1994) The Design and Evolution of C++. Addison-Wesley. Reading, MA.

[VENB98] Venners, Bill (Sep/1998). http://www.javaworld.com/jw-10-1998/jw-10-techniques.html

[WALC77] Walston, C. E., &Felix, C. P. (1977). A Method of Programming Measurements and Estimation. IBM Systems Journal 16, #1: 54-73.

[WAYG94] Wayt Gibbs, Wayt (Sep/1994) Software's Chronic Crisis. Scientific American.http://cispom.boisestate.edu/cis310emaxson/softcris.htm

[WINT96] Winograd, Terry (1996) Bringing Design to Software. Addison Wesley.