Implicaciones de Eiffel en el Diseño Orientado a Objeto

Francisco José García Peñalvo
*Licenciado en Informática. Profesor del Área de Lenguajes y Sistemas Informáticos de la **Universidad de Burgos**.*
fgarcia@.ubu.es

Yania Crespo González-Carvajal
*Licenciada en Ciencias de la Computación por la **Universidad de la Habana**. Becaria del ICI en la **Universidad de Valladolid***
yania@.infor.uva.es

Eiffel intenta romper la visión tradicional de los lenguajes de programación, conjugando características propias del paradigma objetual con conceptos de Ingeniería del Software, de forma que se convierte en algo más que en un simple lenguaje de programación, extendiendo su ámbito de acción para convertirse en un marco de trabajo para el desarrollo de diseños e implementaciones orientadas a objetos

Si antes de leer este artículo, le preguntásemos a los lectores ***¿qué es Eiffel?***, la mayoría de ellos responderían que un lenguaje de programación orientado a objetos (LPOO), y no estarían equivocados. Pero, el objetivo del presente artículo es cambiar la visión del lector sobre Eiffel, de forma que cuando se les repita la pregunta, puedan dar una respuesta que vaya más allá de lo más evidente, reflejando las características que lo convierten en un entorno de desarrollo de software que combina un método, un lenguaje y una biblioteca de clases bajo el paradigma de la orientación al objeto.

Normalmente, cuando se hace referencia a un lenguaje de programación, con independencia del paradigma en el que se encuadre, se está pensando exclusivamente en la herramienta que va a dar soporte a la fase de codificación del software. No obstante, Eiffel puede considerarse como una excepción a esta norma, ya que por aunar en su concepción todas las características propias de la orientación a objetos más elementos de Ingeniería del Software, se posibilita su utilización en fases del ciclo de desarrollo de mayor abstracción que la fase de implementación, más concretamente, es especialmente adecuado para especificar, con un alto nivel de detalle, las clases identificadas en las fases de análisis y diseño preliminar.

Pero, antes de introducirnos en la explicación de los elementos que hacen de Eiffel un excelente lenguaje de especificación de diseños, vamos a hacer un breve repaso de sus características más generales como LPOO.

Introducción a Eiffel como LPOO

Eiffel [1] es un LPOO puro, es decir, un lenguaje diseñado para soportar únicamente el paradigma de la orientación a objeto[1], que fue creado por el Dr. Bertrand Meyer a mediados de la década de los ochenta.

En el diseño de Eiffel, Meyer consigue dotar a su lenguaje de todos los elementos básicos de la orientación al objeto, pero además incluye un conjunto de características

[1] En contraposición con los denominados lenguajes híbridos, como por ejemplo C++, que son extensiones de lenguajes concebidos bajo la influencia de otros paradigmas de programación para soportar las características de la orientación a objeto.

propias de la Ingeniería del Software (*precondiciones, postcondiciones, invariantes*) que le permiten definir un método de diseño, el Diseño por Contrato, una de las grandes aportaciones de Eiffel a la Ingeniería del Software.

Como lenguaje de programación está inspirado en Simula'67, como otros muchos LPOO, aunque recibe influencias de otros importantes lenguajes como son Ada y especialmente Smalltalk.

Eiffel presenta comprobación estricta de tipos, esto es, no puede llamarse a una operación sobre un objeto a menos que en la clase o la superclase de dicho objeto esté definido el prototipo exacto de la operación. Esta característica permite a Eiffel detectar en tiempo de compilación cualquier violación a la concordancia de tipos.

Como todos los LPOO, la abstracción de datos está soportada mediante el uso de clases, permitiéndose tanto la herencia simple como la herencia múltiple entre clases. Otra propiedad de Eiffel es su soporte para la genericidad, es decir, permite definir clases genéricas que, definidas mediante parámetros genéricos formales, representan plantillas de tipos que se derivan en tipos concretos cuando los parámetros genéricos formales se instancian con parámetros genéricos actuales[2].

Otras características a destacar de Eiffel son su soporte de la ligadura dinámica y el polimorfismo, la gestión de memoria automática con recolección de basura (*Garbage Collection*) y la presencia de una importante biblioteca de clases predefinidas en el lenguaje que incluye soporte de diferentes estructuras de datos (*árboles, pilas, colas…*), rutinas matemáticas, manejo de cadenas…

Abstracción	*Variables de instancia* *Métodos de instancia* *Variables de clase* *Métodos de clase*	Sí Sí No No
Encapsulamiento	*De variables* *De métodos*	Público sólo lectura, Privado Público, Privado
Modularidad	*Tipos de módulo*	Clase
Jerarquía	*Herencia* *Unidades genéricas* *Metaclases*	Múltiple Sí No
Tipos	*Comprobación estricta* *Polimorfismo*	Sí Sí
Concurrencia	*Multitarea*	No
Persistencia	*Objetos persistentes*	Sí

Tabla A. Características de Eiffel como LPOO

Estructura de una clase Eiffel

Una clase Eiffel puede dividirse de forma general en: *cabecera, recursos* e *invariantes*. Esto no es más que una manera de agrupar las construcciones lingüísticas correspondientes a la clase con fines didácticos pues una clase Eiffel es un elemento indivisible, al contrario de C++ donde la especificación de la clase puede dividirse en declaraciones y definiciones (*implementación*), llegando incluso a disgregarse en un fichero cabecera que contiene la declaración (*.h*) y en un fichero que contiene las definiciones de los métodos de la clase (*.cpp*).

[2] Supóngase una pila definida en función de un tipo genérico T, de forma que puede servir para hacer funciones de pila de enteros, de caracteres… en función de cómo se instancie el parámetro genérico formal T en un parámetro genérico actual como puede ser INTEGER o CHARACTER.

En la cabecera de una clase Eiffel se inscriben sus declaraciones generales: nombre, especificaciones de indexación, cláusula de herencia, cláusula de creación, etc. La segunda parte, que hemos nombrado recursos, constituye una declaración y definición (*si procede*) de los recursos propios de la clase. La unificación en un mismo concepto de atributos y métodos, donde estos últimos pueden ser de tipo función (*retornan un resultado*) o de tipo rutina, es lo que se denomina en Eiffel recursos (*features*) de una clase. En la última parte, como su nombre lo indica, se especifican las expresiones lógicas que constituyen los invariantes de la clase, es decir, las propiedades que un objeto construido a partir de dicha clase debe conservar a lo largo de todo su ciclo de vida.

Como primer elemento de la cabecera de una clase se tiene la cláusula de indexación, que se identifica sintácticamente por la palabra clave **index**, en la que se agrupan un conjunto de pares etiqueta-valor. Las etiquetas más habituales que pueden ser incluidas son aquellas que indican los sinónimos (**synonyms**), palabras claves (**keywords**) y dominios de aplicación (**domains**) de la clase. La especificación de esta cláusula es completamente opcional, al igual que los pares etiqueta-valor. Ahora bien, la existencia de esta cláusula como parte de la sintaxis de la estructura de una clase es un hecho importante. La información que aparece en la cláusula de indexación está presente por defecto en cualquiera de las formas standard de recuperación de información de clases Eiffel pre-compiladas (*forma short, forma flat [2], [3]*). Además, constituye un punto de partida para las herramientas de localización de clases en repositorios o bibliotecas.

A continuación de la cláusula de indexación (*que al ser opcional puede no aparecer*) se tiene el nombre de la clase, por ejemplo MATRIZ, que puede venir acompañado de la indicación de si esta clase es genérica y, por consiguiente, de la declaración de los parámetros genéricos formales con sus restricciones, MATRIZ[T→NUMERIC]. Esto indica que la clase MATRIZ es una clase genérica con un único parámetro genérico formal identificado por T. La restricción que viene señalada por →, indica que el tipo real con el que se instancie el genérico formal T tiene que conformar con el tipo NUMERIC. La conformancia de tipos en Eiffel está basada principalmente en la herencia de clases (*aunque en realidad la regla es más compleja debido a la genericidad [1]*). NUMERIC es una clase de la jerarquía de clases predefinida de Eiffel de la cual descienden algunos tipos básicos como son INTEGER, REAL... y que define básicamente a un elemento que tiene las operaciones aritméticas (*suma, multiplicación...*). El programador puede crear un elemento numérico propio heredando de NUMERIC, definiendo dichas operaciones como puede ser el caso de una clase COMPLEJO que describa los elementos del campo de los números complejos.

En la cláusula de herencia, que se identifica sintácticamente por la palabra clave **inherit**, se listan los ancestros directos de la clase. Para cada padre pueden incluirse diferentes subcláusulas que especifican, si es el caso, qué recursos heredados se renombran y cómo (*cláusula **rename***), qué recursos heredados se redefinen (*cláusula **redefine***), qué métodos pierden su definición, con esto se consigue que la clase pase a ser abstracta porque no se aporta definición para dicho recurso (*cláusula **undefine***) y qué recursos cambiarán su exportación y cómo (*cláusula **export***), entre otras.

Entre los métodos que tiene una clase se puede especificar cuál o cuáles de ellos constituyen métodos de creación. Esto se hace a través de la cláusula de creación, identificada por la palabra clave **creation**, que es una lista de uno o más métodos. Los métodos que se declaran en dicha cláusula son los que pueden aparecer como parte de una instrucción de creación de objetos de la clase. Las acciones más comunes que se

realizan en estos métodos pasan por la iniciación de los atributos del objeto de modo que desde su nacimiento satisfagan los invariantes de la clase. Un método puede exportarse como método de creación y como otro más de los recursos de la clase. Si este es el caso, además de poderse utilizar como parte de una instrucción de creación, puede ser invocado en cualquier momento de la vida del objeto (*como un método más*). En cambio, también es posible especificar que un método se utilice sólo con fines de creación. Esto se hace incluyéndolo en la cláusula de creación y, por otra parte indicando que no se exporte como recurso (*en la parte correspondiente a los recursos*).

El otro gran grupo, en el que hemos dividido la estructura de una clase Eiffel, corresponde a los recursos. La palabra clave **feature** identifica el comienzo de una sección en la que se definirán uno o más recursos de la clase. Cada grupo de recursos (*marcado por la palabra* ***feature***) puede ir acompañado de una especificación de exportación. Los recursos pueden exportarse a todas, a ninguna o a una o más clases. Si no se especifica nada, por defecto, se asume la exportación a {ANY}, es decir a todas las clases. Para no exportar a ninguna clase se declara {NONE}. Para exportar los recursos a una o varias clases se lista entre llaves el nombre de éstas.

Como parte de las especificaciones de recursos, se definen atributos variables o constantes y métodos rutinas o funciones. La definición de un atributo variable consiste en especificar su nombre y su tipo, mientras que la de un atributo constante en indicar nombre, tipo y valor. La especificación de un método empieza por indicar si es diferido (virtual) o no y declarar su signatura (*nombre, nombre y tipo de los parámetros y tipo del resultado para el caso de las funciones*). Como parte de la declaración del método, independientemente de si es o no diferido, se pueden especificar sus precondiciones y postcondiciones. Si el método no es diferido se acompaña de su definición, es decir, la secuencia de instrucciones que constituyen su implementación.

En la parte correspondiente a los recursos también aparecerá la nueva definición que la clase actual da para aquellos recursos heredados que se redefinirán, si así se especifica en la subcláusula **redefine** de la cláusula **inherit**. Una redefinición puede ser un cambio del tipo de un atributo, de un parámetro de un método o del resultado de una función. El nuevo tipo deberá conformar con el anterior (*redefinición covariante*). También, una función sin parámetros puede redefinirse, cuando se hereda, como un atributo (*pero no viceversa*). Por ejemplo, si se tiene una clase PERSONA con una función que retorna la edad, calculada a partir de la fecha de nacimiento, en una clase heredera puede redefinirse la función edad como un atributo. Dicha redefinición puede ir acompañada de incluir otro método en la clase que incremente la edad el día del cumpleaños de la persona. Por último, en general, un método puede redefinirse proporcionando otra implementación y en consecuencia, las precondiciones y postcondiciones podrán cambiarse también, pero no libremente [1], [2]. Ni los atributos constantes, ni los recursos declarados como **frozen** pueden redefinirse.

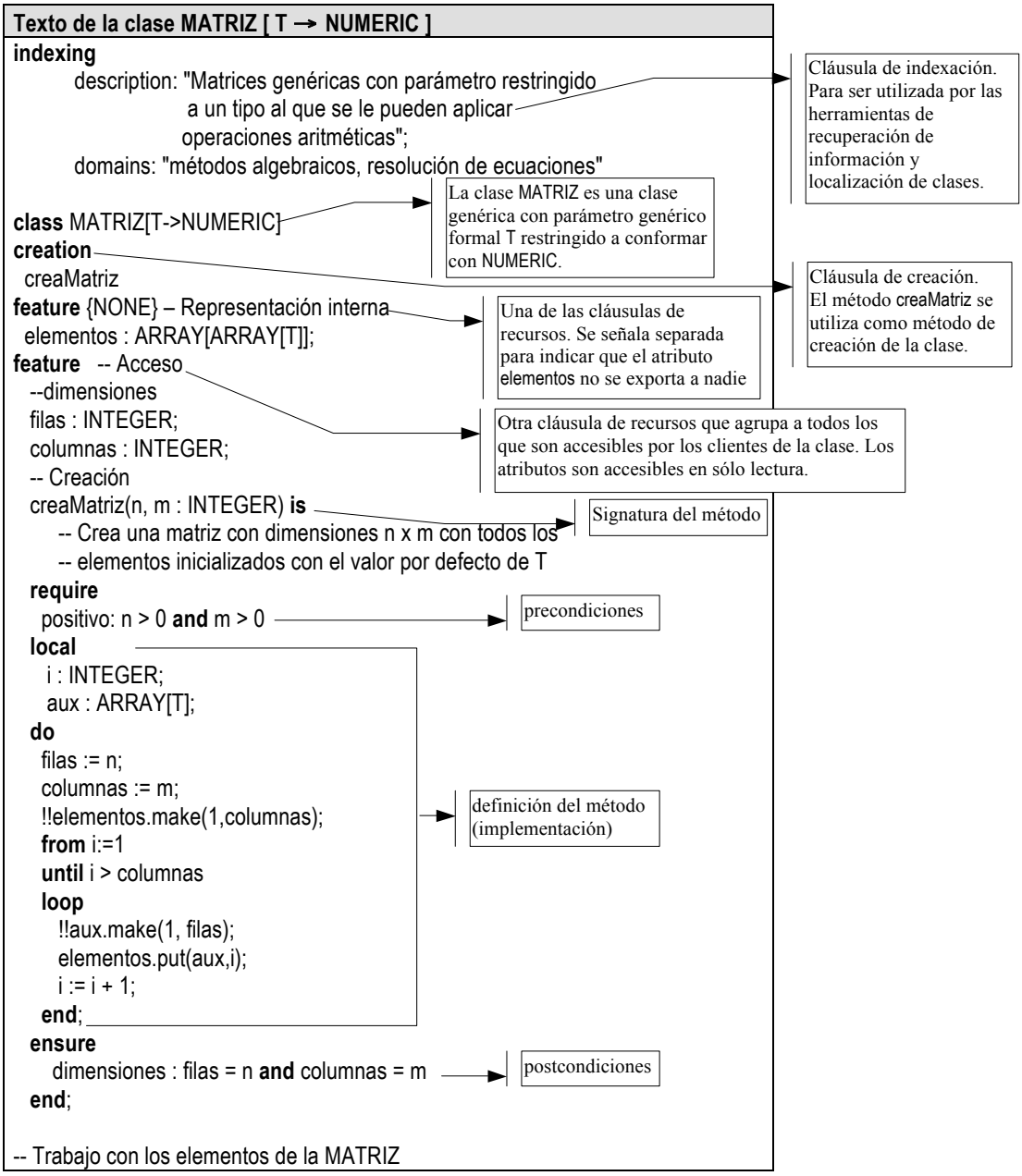
La que se ha clasificado como última parte de la estructura de una clase, es la que corresponde a la definición de sus invariantes. Esta sección se identifica sintácticamente con la palabra clave **invariant**. La especificación de los invariantes no es más que declarar un conjunto de expresiones lógicas en las que están involucrados los recursos de la clase. Con la herencia pueden añadirse nuevos invariantes para la nueva clase, pero tienen que respetarse los que se indican en los ancestros.

En el **Listado 1** se presenta un ejemplo de una clase Eiffel.

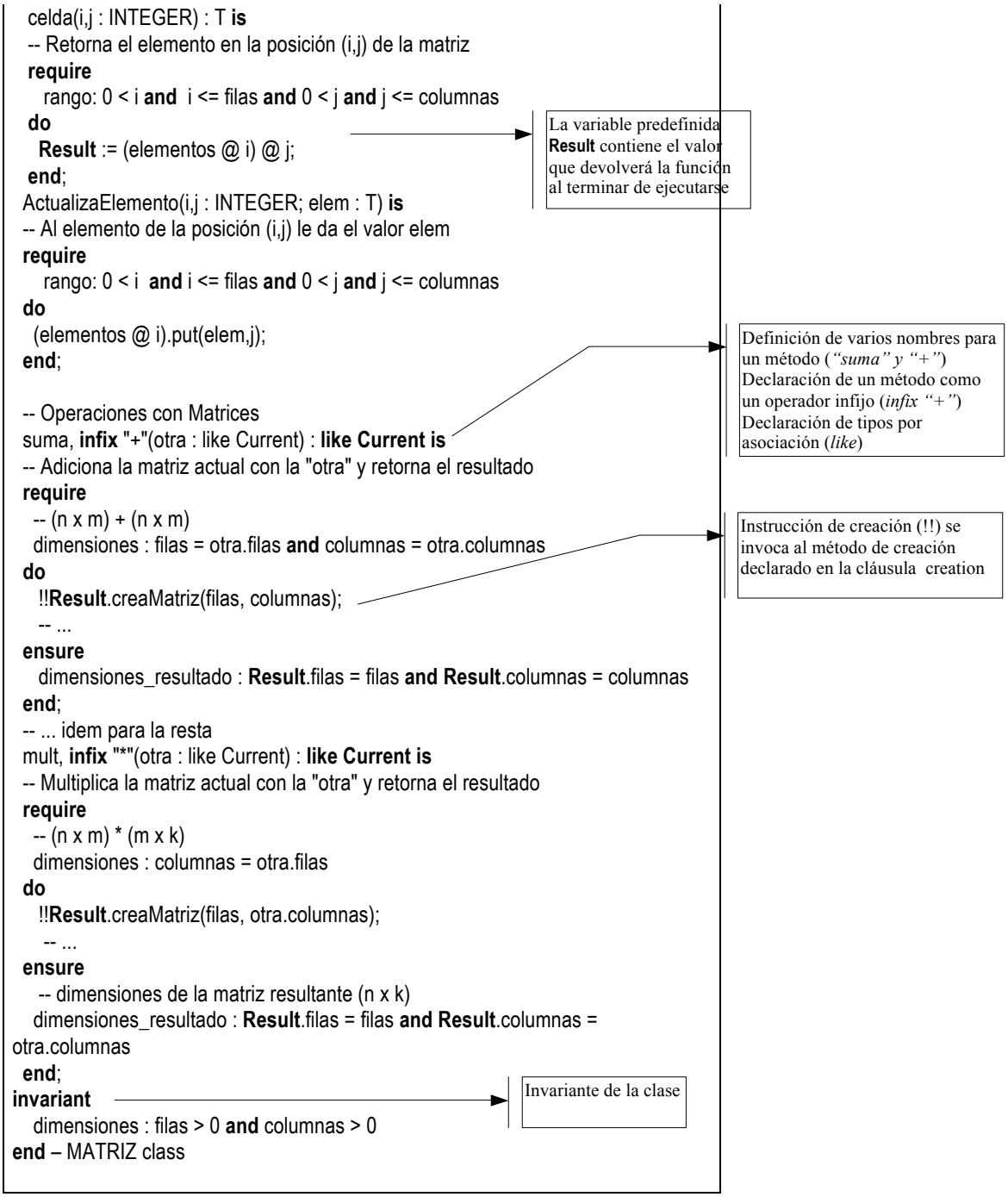
Entornos de desarrollo basados en Eiffel

Los entornos de desarrollo basados en Eiffel más conocidos son los entornos de ISE Eiffel, Tower Eiffel y Small Eiffel.

ISE (*Interactive Software Engineering*)[3] ha construido un entorno de trabajo llamado Eiffel Bench tanto para Windows95/NT como para plataformas de estaciones de trabajo con interfaces gráficas basadas en Motif. ISE Eiffel es un compilador y entorno de trabajo que incluye bibliotecas de clases reutilizables, herramientas para manejar las clases, el sistema, herramientas de recuperación de la información de clases, de análisis y diseño de clases (EiffelCASE) y de ingeniería inversa. Consta de un intérprete que permite la ejecución del sistema, la depuración simbólica... La arquitectura del compilador está basada en la tecnología **Melting Ice** [2 pp. 1145]. Por otra parte, permite la generación de aplicaciones ejecutables independientemente del entorno, mediante un enlace con los compiladores de C++: Visual C++ o Borland C++ 5.0.



[3] http://www.eiffel.com



Listado 1. Ejemplo de la clase MATRIZ en Eiffel.

Small Eiffel es un compilador de Eiffel de libre distribución para UNIX, DOS, OS/2 y con versiones *stand-alone* para Macintosh desarrollado por un grupo de investigación de Loria[4]. Small Eiffel es un compilador que genera código C y se integra con GNU-C para obtener las aplicaciones ejecutables.

Por su parte, Tower Eiffel es también un compilador y entorno de trabajo desarrollado por Tower Technology[5]. Tower Eiffel está disponible para Windows95/NT, SPARC Solaris y SunOS, IBM AIX, SGI Irix, x86Linux... Incluye además del compilador y el ambiente de programación otras prestaciones como, por ejemplo, generación automática

[4] http://www.loria.fr/SmallEiffel
ftp://ftp.loria.fr/pub/loria/genielog/SmallEiffel
[5] http://www.twr.com/eiffel/eiffel-products.html

de documentación de clases, biblioteca de clases reutilizables, depurador a nivel de código fuente...

Influencia de Eiffel en el Diseño Orientado a Objeto

Como se ha puesto de manifiesto en apartados anteriores, Eiffel pretende ser más un entorno de Ingeniería del Software que un LPOO, siendo perfectamente aplicable como lenguaje de especificación de diseños orientados a objeto, como se puede comprobar en el **Listado 1**. En consecuencia, nos vamos a centrar en el aspecto fundamental con el que Eiffel ha incidido en la tecnología de objetos en particular y en la Ingeniería del Software en general: el *diseño por contrato* [2].

El diseño por contrato es el corazón del método que se encuentra tras el diseño de Eiffel, y que consiste en la aplicación de un conjunto de técnicas sistemáticas para construir software fiable, extensible y reutilizable. Mediante esté método de diseño el sistema software se concibe como un conjunto de componentes que se comunican de una forma precisa estableciendo obligaciones y beneficios mutuos, esto es, los contratos. La forma que tiene Eiffel de dar soporte a esta filosofía de desarrollo es mediante aserciones.

Eiffel incita a los desarrolladores de software a expresar las propiedades formales de las clases mediante el uso de aserciones, las cuales pueden presentar tres formas concretas: precondiciones, postcondiciones e invariantes.

Las precondiciones expresan los requisitos que un cliente debe satisfacer cuando éste invoca un servicio y se introducen en Eiffel utilizando la cláusula **require**.

Las postcondiciones expresan las condiciones que el servicio debe garantizar al terminar su operación, siempre que la precondición se satisfaciera en la entrada del servicio. Las postcondiciones se expresan en Eiffel mediante el uso de la cláusula **ensure**.

Las invariantes de una clase es una condición que cualquier instancia de la clase debe satisfacer en todo circunstancia estable. Representan, por tanto, una restricción de consistencia impuesta a todos los servicios de la clase. Las invariantes de la clase aparecen en la cláusula **invariant**.

Sintácticamente, las aserciones son expresiones booleanas del cálculo proposicional, es decir, sin cuantificadores, aunque ya se han propuesto extensiones al lenguaje para soportar cálculo de predicados (*con cuantificador universal y existencial*). Como parte de las expresiones booleanas pueden utilizarse llamadas a funciones que retornan **BOOLEAN** definidas en las clases del sistema. Una característica adicional es la introducción de la palabra clave **old** que se utiliza como calificativo de un atributo para indicar el valor que tenía el atributo antes de entrar a la rutina. Por este motivo, la expresión **old** sólo puede aparecer en una postcondición (**Listado 2**).

Las aserciones de Eiffel sirven para exponer de forma explícita en la definición de las clases aquellas afirmaciones en las que los programadores confían cuando crean los sistemas software que ellos consideran correctos.

La idea de contrato establece una relación contractual entre un servicio y el cliente que lo invoca, utilizando las precondiciones y postcondiciones para expresarlo, de forma que el cliente debe asegurar que se cumple la precondición del servicio que va a invocar, y el servicio (*localizado en el servidor*) asegura que después del mismo las postcondiciones se cumplen. Esto lleva a la existencia de una confianza mutua entre cliente y servidor que se hace explícita gracias a las aserciones.

BON: Una metodología de Ingeniería de Software Orientada a Objeto

BON es una metodología que se inscribe en la categoría de metodologías puramente orientadas al objeto. Tiene su origen en la filosofía de diseño y programación por contrato [2], junto con algunas características del modelo de objetos propuesto por Eiffel [1] y diversas influencias del proyecto europeo ESPRIT II (*Research and Development Program*).

```
class Cuenta
feature
    balance: INTEGER
    propietario: PERSONA
    balance_minimo: INTEGER

    abrir(quien: PERSONA) is
        do
            propietario := quien
        end

    deposito(sum: INTEGER) is
        require
            sum >= 0
        do
            add(sum)
        ensure
            balance = old balance + sum
        end

    -- Resto de los servicios de la clase Cuenta

feature {NONE}
    add(sum: INTEGER) is
        do
            balance := balance + sum
        end
end -- clase Cuenta
```

Listado 2. Clase Cuenta

BON es un acrónimo que significa Business Object Notation. Como metodología, aunque generalmente aparece ligada a Eiffel, su concepción es independiente de este lenguaje pero indiscutiblemente muchas de sus técnicas encuentran un equivalente inmediato en Eiffel en el proceso de implementación. Esto hace que se facilite la construcción de herramientas CASE orientadas a este lenguaje en particular ISE Eiffel, por ejemplo, soporta BON con la herramienta EiffelCASE.

BON es un método y una notación para análisis y diseño orientados a objeto de sistemas de alto nivel, que hace énfasis en permitir un desarrollo sin transiciones[6], en hacer posible la reutilización del software a gran escala y en reflejar la confianza necesaria para hacer que los componentes reutilizables se acepten y utilicen por la industria del software.

Sus principios fundamentales son:

- *Simplicidad*
- *Desarrollo "sin costuras"*

[6] Continuo, sin costuras, sin transición; del término inglés seamless.

- *Software contractual*
- *Reversibilidad*
- *Escalabilidad*

El desarrollo sin costuras es el principio de utilizar un conjunto consistente de conceptos y notaciones a través del ciclo de vida del software, evitando los problemas de impedancia de los métodos tradicionales[7]. Una ventaja de mantener este principio viene del hecho de que los principales esfuerzos en el desarrollo del software se emplean, no en nuevos desarrollos sino, en mantener el software. Una metodología en la que no se encuentre un salto entre las diferentes etapas está preparada para reflejar con mayor facilidad y claridad el proceso de cambio. Otra ventaja muy importante del *seamless* es que facilita los procesos de traducción automática. Las metodologías que promueven un proceso de desarrollo sin costuras se ven más como el soporte intelectual necesario a través del proceso completo de construcción del software que como la especificación de cada una de las etapas separadas del ciclo de vida del software. Unido a esto, está la presencia del principio de simplicidad, esto es, este principio indica que hay que minimizar el número de conceptos utilizados.

En cuanto al principio de software contractual ya ha quedado explicado en el apartado anterior.

El principio de reversibilidad complementa el proceso sin costuras, garantizando que los cambios realizados en cualquier paso del proceso, incluso en la implementación o el mantenimiento, puedan reflejarse hacia atrás en los primeros pasos. De esta manera, se garantiza el estado consistente del proyecto, lo que significa que la continuidad debe funcionar en ambas direcciones. Si esto no fuera posible, o demasiado difícil de realizar en la práctica, los primeros niveles de modelado pasan a ser obsoletos dejando solamente el código fuente de la implementación como especificación del sistema, con todos los inconvenientes que esto supone.

El principio de escalabilidad se manifiesta en términos de la capacidad de soportar un formalismo para representar grupos de clases progresivos y soportar la división del problema basado en capas de abstracción como manejo de la complejidad estructural.

En un método de análisis y diseño orientado al objeto sin costuras, la continuidad entre los pasos conduce a la conclusión de que no existe una clara distinción entre qué es lo que realmente pertenece al análisis y qué pertenece al diseño. En realidad, la idea de BON es caracterizar los objetivos y procesos específicos del análisis y del diseño pero asegurando que el paso de uno a otro no esté regido por ninguna condición, que las notaciones y conceptos que se emplean en cada una de las etapas sean consistentes en el sentido de que el mismo concepto solamente evoluciona en la medida que se avanza, y que apoyándose en la reversibilidad, se garantice la iteratividad en el ciclo de vida del software.

El objetivo del análisis es poner un cierto orden en nuestra concepción del mundo real. El propósito es simplificar, dominar la complejidad mediante la reformulación del problema. En el análisis deben eliminarse redundancias en las especificaciones, ruidos; encontrar inconsistencias, posponer decisiones de implementación, dividir el espacio del problema, tomar un cierto punto de vista y documentarlo. El método BON considera que el análisis se convierte en diseño cuando se toman decisiones de implementación, cuando se introducen grados de protección para la información o cuando se introducen

[7] Lo que en la literatura inglesa se conoce con el término de *Impedance Mismatches*.

clases que no están relacionadas con el espacio de objetos del problema. El diseño es un proceso que toma por entrada una representación del problema y la transforma en una representación de la solución, regresando al análisis si es necesario. En el diseño, las clases obtenidas en el análisis se extienden, generalizan y se transforma su representación en un esquema fácilmente traducible a un lenguaje de programación orientado al objeto. Se añade la especificación de nuevas clases que aparecen como interfaz con el exterior del sistema, otras que son de utilidad básica para la construcción del sistema (*clases contenedoras*...) y aquellas que son consideradas como clases de aplicación que tratan con información dependiente de la máquina o del sistema, con persistencia de objetos, manejo y recuperación de errores.

En BON se modelan tanto el espacio del problema como el espacio de la solución como abstracciones de datos que encapsulan servicios. En el modelado, aunque lo primero en que se piense sea en un servicio, debe buscarse la abstracción (*clase*) subyacente que representa a aquellos individuos que ofrecen tal servicio. El primer principio es considerar las clases como una representación de tipos de datos abstractos que tienen un estado interno y ofrecen servicios, opuesto al criterio "las cosas que hacen" que está más cerca de las técnicas procedurales.

Las clases no se ubican aisladamente sino que se agrupan en clusters. Durante el análisis, los clusters ayudan en cuanto agrupan clases de acuerdo con un criterio de proximidad basado en funcionalidad de subsistema, en nivel de abstracción o en el punto de vista del usuario final. En el diseño, son utilizados frecuentemente como una técnica estructurada para visualizar selectivamente las conexiones entre las clases. Es muy común comenzar por un cluster general y luego ir determinando otros después de haber hecho ciertos agrupamientos de clases. Es importante ubicar el lugar de una clase dentro de la estructura completa. Esto implica que encontrar clases relacionadas es más significativo que encontrar una clase aislada. El uso de los clusters es una técnica para dotar de escalabilidad al método.

Cada etapa, análisis y diseño, en BON aporta al modelo general desde dos puntos de vista distintos: arquitectura y comportamiento. El resultado es un modelo subdividido en modelo estático y modelo dinámico. Las actividades globales que describe el método para ir creando dichos modelos se describen en la **Tabla B** mientras que en la **Tabla C** y en la **Figura A** se describen los elementos gráficos más generales que se utilizan en la descripción del modelo.

Es de destacar, que como una tarea fundamental en el método, de considerar completo el análisis y diseño y pasar a la fase de implementación, se propone comenzar un proceso de generalización. En este proceso debe analizarse si la arquitectura del sistema es suficientemente general para maximizar futuras reutilizaciones. De esta manera, del conjunto de clases obtenidas se factorizan recursos comunes en clases de alto nivel (*por herencia o por genericidad*) y las clases existentes se convierten en versiones especializadas de estas últimas.

Conclusiones

En el presente artículo se ha querido mostrar la relevancia que para el proceso de diseño tiene un lenguaje como Eiffel, no tan conocido como C++, pero con unas características que lo convierten para muchos en el LPOO más completo que existe.

De todos los conceptos presentados, el diseño por contrato supone un importante hito en la Ingeniería del Software, lo cual se ve reflejado de una u otra forma en diferentes

metodologías, de las cuales el ejemplo más relevante es BON, método orientado a objetos que también ha sido presentado en este artículo.

TAREA	DESCRIPCIÓN	ESQUEMA
1. Encontrar clases	Delinear la frontera del sistema. Encontrar subsistemas, metáforas de los usuarios, casos de uso.	Tabla de Sistema (System Chart), Tablas de Escenarios (scenario Charts)
2. Clasificar	Listar clases candidatas. Crear un glosario de términos técnicos.	Tabla de Clusters (Clusters Chart)
3. Encontrar clusters	Seleccionar clases y agruparlas en clusters. Clasificar, esbozar colaboraciones fundamentales entre clases	Tabla de Sistema (System Chart), Tabla de Clusters (Clusters Chart), Arquitectura (Static Architecture), Diccionario de clases (Class Dictionary)
4. Definir los recursos de las clases	Definir las clases. Determinar comandos (¿Qué servicios pueden solicitar otras clases a esta?), consultas (¿Qué información pueden preguntar otras clases a esta?) y restricciones (¿Qué conocimiento debe mantener la clase?)	Tablas de Clases (Class Charts)
5. Seleccionar y describir escenarios de objetos	Esbozar el comportamiento del sistema. Identificar eventos, creación de objetos y escenarios relevantes derivados de la utilización del sistema	Tablas de Eventos (Event Charts), Tablas de Escenarios (Scenario Charts), Tablas de Creación (Creatio Charts), Escenarios de Objetos (Object Scenarios)
6. Especificación de las condiciones contractuales.	Definir los recursos públicos. Especificar tipos, signaturas y contratos formales.	Interfaz de clases (Class Interfaces), Arquitectura (Static Architecture)
7. Refinar el sistema.	Encontrar nuevas clases de diseño, adicionar nuevos recursos.	Interfaz de clases (Class Interfaces), Arquitectura (Static Architecture), Diccionario de clases (Class Dictionary), Tablas de Eventos (Event Charts), Escenarios de Objetos (Object Scenarios)
8. Incrementar el potencial de reusabilidad	Generalizar. Factorizar comportamiento común.	Interfaz de clases (Class Interfaces), Arquitectura (Static Architecture), Diccionario de clases (Class Dictionary)
9. Indexar y Documentar	Documentar explícitamente la descripción de una clase mediante la cláusula de indexado. Completar la información de documentación en el diccionario de clases.	Interfaz de clases (Class Interfaces) Diccionario de clases (Class Dictionary)
10. Evolucionar la arquitectura del sistema	Completar y revisar el sistema. Producir una arquitectura final con un comportamiento del sistema.	Modelos estático y dinámico finales de BON. Todos los esquemas completados.

Tabla B. Actividades globales de BON.

Tabla de Sistema (System Chart): Definición del sistema y lista de los clusters asociados. Solamente una Tabla de sistema por proyecto. Los subsistemas se describen mediante su correspondiente Tabla de Cluster.
Tablas de Clusters (Cluster Charts): Definición de clusters y lista de las clases asociadas y subclusters, si los hay. Un cluster representa un subsistema completo ó sólo un grupo de clases.
Tablas de Clases (Class Charts): Definición de las clases de análisis en términos de comandos, consultas y restricciones, de forma que sea entendible para los expertos en el dominio y personal no técnico.
Diccionario de Clases (Class Dictionary): Una lista alfabéticamente ordenada de todas las clases del sistema, mostrando el cluster al que pertenece cada clase y una breve descripción. Debe poder ser generada automáticamente de las Tablas de Clase y de Interfaz.
Arquitectura (Static Architecture): Conjunto de diagramas que posiblemente representan clusters anidados, encabezamientos de clases y sus relaciones. Una vista del sistema (con posibilidades de hacer zoom)
Interfaz de Clases (Class Interfaces): Definiciones tipadas de las clases con la signatura de los recursos y contratos formales con un lenguaje basado en el cálculo de predicados. Vista detallada del sistema
Tablas de Creación (Creation Charts): Lista de las clases que están a cargo de crear instancias de otras clases. Normalmente se hace una para el sistema pero si se desea se puede incluir una por subsistema.
Tablas de Eventos (Event Charts): Conjunto de eventos externos (estímulos) que disparan algún comportamiento interesante del sistema y el conjunto de respuestas del sistema. Puede ser repetido para cada subsistema.
Tablas de Escenarios (Scenario Charts): Lista de los escenarios de objetos utilizados para mostrar algún comportamiento interesante y representativo del sistema. Los subsistemas pueden contener Tablas de Escenario locales.
Escenarios de Objetos (Object Scenarios): Diagramas dinámicos que muestran comunicaciones relevantes entre objetos para algunos o todos los escenarios que se describen en las Tablas de Escenarios.

Tabla C. Elementos más característicos de BON.

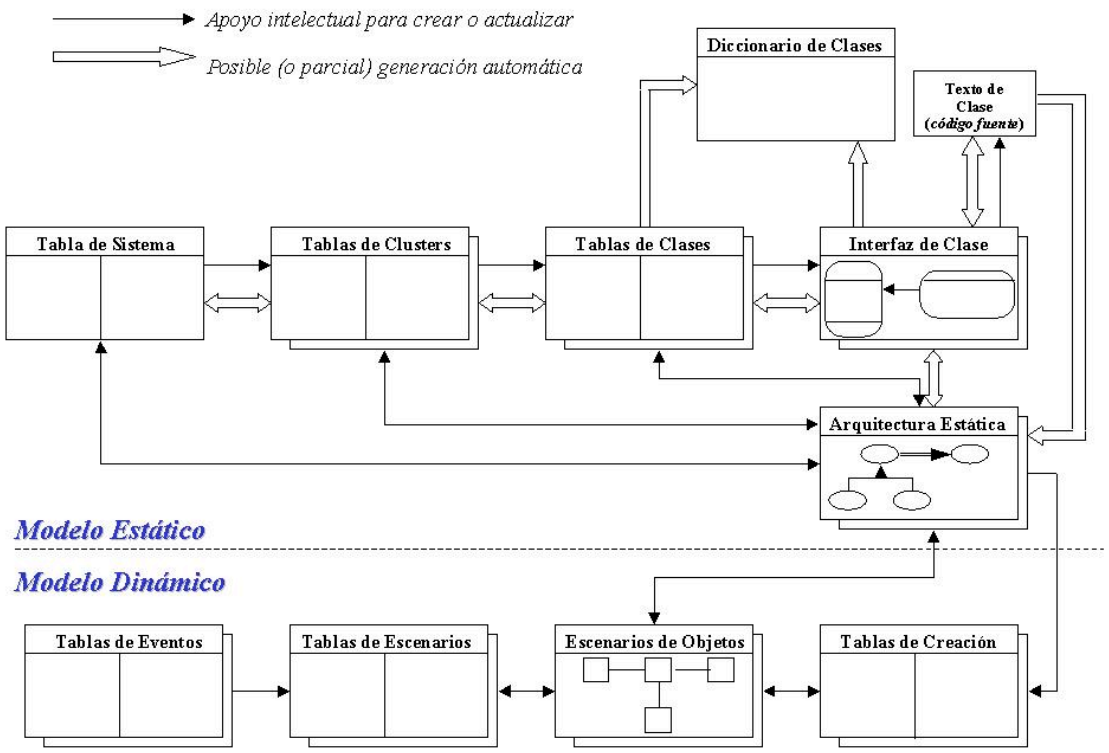


Figura A. Elementos gráficos de BON

Referencias

[1] Meyer, B. "*Eiffel: The language*". Prentice Hall, 1992

[2] Meyer, B. "*Object Oriented Software Construction*". Second Edition, Prentice Hall International 1997.

[3] Meyer, B. "*Tools for a New Culture: Lessons from the Design of the Eiffel Libraries*". Communications of the ACM vol. 33, nº 9, Sept. 1990.

[4] Nerson, J-M. "*Applying Object-Oriented Analysis and Design*". Communications of the ACM vol. 35, Nº 9, Sept. 1992.

[5] Waldén, K. & Nerson, J-M. "*Seamless Object-Oriented Software Architecture, Analysis and Design of Reliable Systems*". Object-Oriented Series, Prentice Hall International, 1995.