DISEÑO POR CONTRATOS: CONSTRUYENDO SOFTWARE CONFIABLE

Gerardo Rossel, grossel@dc.uba.ar
Andrea Manna, amanna@dc.uba.ar

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

DISEÑO POR CONTRATOS: CONSTRUYENDO SOFTWARE CONFIABLE

Gerardo Rossel [1], Andrea Manna[2]
grossel@dc.uba.ar , amanna@dc.uba.ar
[1]Facultad de Tecnología Informática.
Universidad Abierta Interamericana. Argentina
[1, 2] Departamento de Computación.
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires. Argentina

Resumen La construcción de software confiable es uno de los desafíos de la Ingeniería de Software. En este trabajo se presentan los conceptos principales del diseño por contratos. Las técnicas del diseño por contratos afectan todas las actividades del desarrollo de software: desde el análisis hasta la implementación: corrección, reuso, depuración, testeo, documentación y administración. Si bien el lenguaje de programación Eiffel es el lenguaje comercial que actualmente tiene el soporte nativo para contratos, sus ideas y conceptos pueden ser aplicados en diferentes lenguajes (JAVA, C++, C#, etc.).

Palabras clave. Diseño por Contratos, Software Orientado a Objetos, Ingeniería de Software.

DESIGN BY CONTRACT: PRODUCING RELIABLE SOFTWARE

Gerardo Rossel[1] , Andrea Manna[2]
grossel@dc.uba.ar , amanna@dc.uba.ar
[1]Facultad de Tecnología Informática.
Universidad Abierta Interamericana. Argentina
[1, 2] Departamento de Computación.
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires. Argentina

Abstract. The construction of reliable software is one of the software engineering challenges. In this paper we present the main concepts of Design by Contract. The techniques of design by contracts affect all software development activities: from analysis to implementation: correction, software reuse, debugging, testing, documentation and management. Although the programming language Eiffel is the commercial language that at the moment has the native support for contracts, the ideas and concepts can be applied in different languages. (JAVA, C++, C#, etc.).

Keywords. Design By Contract, Object Oriented Software , Software Engineering

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

INTRODUCCIÓN

Las ideas del Diseño por Contratos (Bertrand, 1992, 1997 y Mitchell, 2002), tienen sus raíces en los métodos formales para la construcción de software, pero mantienen una visión más pragmática. Requieren muy poco esfuerzo extra pero generan software mucho más confiable. La importancia de comprender el diseño por contratos, queda clara en la afirmación del pionero de la ingeniería de software Tom DeMarco: *"Yo creo que el uso de contratos a la Eiffel entre módulos es la no-práctica más importante en el mundo del software hoy. Por esto yo quiero decir que no hay ninguna otra práctica siendo impulsada actualmente que tenga mayor capacidad de mejorar la calidad del software producido."* (DeMarco 1997).

El diseño por contratos puede ser visto como la aplicación a la construcción de software de los contratos que rigen los asuntos de las personas. Cuando dos personas establecen un contrato se desprenden, de éste, las obligaciones y beneficios de cada una. Este tipo de contratos en software especifican, en forma no ambigua, las relaciones entre las rutinas y los llamadores de las mismas. Podemos ver un sistema como un conjunto de elementos de software interrelacionados, donde cada uno de los elementos tiene un objetivo con vistas a satisfacer las necesidades de los otros. Dichos objetivos son los contratos. Los contratos deben cumplir, por lo menos, con dos propiedades: ser explícitos y formar parte del elemento de software en sí mismo.

El Diseño por Contratos da una visión de la construcción de sistemas como un conjunto de elementos de software cooperando entre sí. Los elementos juegan en determinados momentos alguno de los dos roles principales *proveedores* o *clientes*. La cooperación establece claramente *obligaciones* y *beneficios*, siendo la especificación de estas obligaciones y beneficios los contratos. Un contrato entre dos partes protege a ambas. Por un lado protege al cliente por *especificar cuanto debe ser hecho* y por el otro al proveedor por especificar el *mínimo servicio aceptable.*

Siguiendo la metáfora de los contratos entre las personas, se puede ver el ejemplo propuesto en (Waldén,1995) que es a su vez una modificación del usado en (Meyer,1992.). Si alguien se encuentra en Estocolmo y necesita enviar un paquete importante a una dirección en la otra punta de la ciudad. Hay dos opciones: llevar el paquete uno mismo o enviarlo mediante alguna empresa de mensajería. Si se elige la segunda opción se establece un contrato entre el servicio de mensajería y el que necesita enviar el mensaje. En Estocolmo existe el servicio "Green Delivery" que es prestado mediante bicicleta. El siguiente cuadro muestra un resumen del acuerdo:

	Obligaciones	Beneficios
Proveedor	Proveer de un paquete de un pero menor a 35kg y dimensiones para bulto de 50 x 50 x 50cm y para documentos de 60 x 80 cm. Pagar 100 SEK.	Tener el paquete entregado dentro de los límites de la ciudad en 30 minutos sin preocuparse por el tráfico ni por la lluvia.
Cliente	Enviar el paquete al destinatario en 30 minutos o menos independientemente de las condiciones de tráfico o lluvia.	No necista preocuparse por entregas muy grandes o pesadas o no pagas.

Claramente, las obligaciones de una parte son los beneficios de la otra parte. Una de las reglas más importantes para el diseño de contratos es el principio de ***No Cláusulas Ocultas***. Es decir que, si el contrato establece que el cliente debe cumplir con determinado requisito, ese requisito *es todo* lo que el cliente debe cumplir, o sea, no hay sorpresas. Tómese en cuenta que existen regulaciones externas

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

que forman parte del contexto más general donde se desenvuelve el contrato (el paquete no puede contener una bomba). En el software, estas regulaciones externas, como se verá luego, toman la forma de *invariantes de clase.*

Dado que los ejemplos que se mostrarán son realizados en el lenguaje Eiffel se usará la terminología del mismo (Meyer,1992b): las clases están compuestas de *características* (*features*) las cuales pueden ser *rutinas* (procedimientos o funciones) o *atributos*. En otros contextos la terminología cambia (*funciones miembro y variables miembro en* C++, *métodos y variables de instancia* en Smalltalk)

ASERCIONES PARTE 1

Los contratos de software se especifican mediante la utilización de expresiones lógicas (más una forma de especificar el valor anterior a una computación: ***old***) denominadas *aserciones*. Existen diferentes tipos de aserciones. En el Diseño por Contratos se utilizan tres tipos de aserciones:

- Precondiciones
- Poscondiciones
- Invariante de Clase.

Antes de explicar las aserciones y su rol en el Diseño por Contratos, se detallará el concepto de *tripleta de Hoare* (Hoare 1969) la cual es una notación matemática que viene de la validación formal de programas. Sea **A** alguna computación y ***P***, ***Q*** aserciones, entonces la siguiente expresión:

{ P } A { Q }

representa lo que se llama fórmula de corrección. La semántica de dicha fórmula es la siguiente: cualquier ejecución de **A** que comience en un estado en el cual se cumple ***P*** dará como resultado un estado en el cual se cumple ***Q***. Por ejemplo

{ x > 10 } x := x /2 { x >= 5}

O sea que si se parte de un estado en el cual **x** es mayor que 10 y luego se aplica la computación x := x/2 entonces como resultado x será mayor o igual a 5. A la aserción ***P*** se la llama *precondición* y a ***Q*** se la llama *poscondición*. En el ejemplo la precondición es x > 10 y la poscondición x >=5.

PROGRAMACIÓN DEFENSIVA VS. DISEÑO POR CONTRATOS

El diseño y la programación por contratos es en cierto sentido opuesto a lo que se denomina *programación defensiva* (Meyer,1992). Si bien existen diversas acepciones para el concepto de programación defensiva, se tomará aquí la idea por la cual la programación defensiva incita a los programadores a realizar todas las verificaciones posibles en sus rutinas de tal forma de prevenir que una llamada pueda producir errores. Como resultado, las rutinas deben ser lo más generales posible evitando aquellas que funcionen sólo si se las llama con determinadas condiciones. Si bien esto puede parecer razonable, tiende a aumentar drásticamente la complejidad del código. Como lo plantea Meyer en (Meyer,1992), los chequeos ciegos (aquellos hechos por las dudas) agregan más software por lo cual agregan más lugares donde las cosas pueden andar mal, lo cual da la necesidad de agregar más verificaciones, que a su vez agrega más software,

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

lo cual.....así al infinito. Básicamente la idea de programación defensiva no apunta a que los elementos de software garanticen especificaciones bien conocidas sino que sean textos ejecutables en condiciones arbitrarias. Otro de los problemas (asociado al reuso de software) es que generalmente no queda claro que hacer con lo valores incorrectos. Con un ejemplo se pueden clarificar las cosas. Supóngase una clase **A** que implementa una rutina que a partir de la suma de valores que tiene guardados en un vector y un parámetro entero, devuelve la parte correspondiente (es decir la división de la suma de valores por el parámetro). A continuación, el código en Eiffel:

```
class A
feature
   get_parte( valor: INTEGER): REAL is
   do
      if valor > 0 then
         Result := get_total / valor
      else
          -- Mostrar mensaje de error
      end
  end
  get_total : Real is
  do
         ..........
  end
end --class
```

Por razones de simplicidad, sólo se muestra el código necesario para entender el ejemplo. La rutina *get_parte*, de acuerdo a la programación defensiva, verifica que el parámetro sea mayor que cero para de esta forma poder hacer la división sin problemas. ¿Qué tiene de malo?. Suponiendo que los clientes saben que el parámetro debe ser mayor que cero, necesitan hacer el mismo chequeo. Si no lo saben, se enterarán en tiempo de ejecución. Al llamarla con un valor menor a cero, obtendrán un mensaje de error. Por otro lado ¿cómo puede reusarse la clase en otro contexto? es decir: ¿un mensaje es la mejor solución para un parámetro erróneo en todos los contextos? Si se quiere reusar la clase para una aplicación que corra en modo *background*, el mensaje de error no es la mejor alternativa. Si la clase pertenece a una biblioteca de clases ¿es correcto que interactúe con los usuarios mediante mensajes?. La solución podrá ser que devuelva un código de error , por ejemplo –1, si el llamado es erróneo. Ahora bien ¿dónde se documenta eso?. ¿Y si se retorna una excepción?. En ese caso no hace falta el chequeo ya que la división por cero provocará la excepción. Pero bien ¿dónde está el error que provocó la excepción, en la rutina o en el llamador? Si está en la rutina, se debería corregir y si está en el llamador ¿cómo sabe éste cual es la forma correcta de llamar a la rutina?. La moraleja es que *solamente* el cliente tiene la información necesaria para operar en caso de tener un valor menor que cero.

La solución del Diseño por Contratos, basado en el principio de *no redundancia*, implica que se debe especificar claramente bajo que condiciones debe llamarse a la rutina (mediante las precondiciones) y a partir de ello se establece cual es el resultado (las poscondiciones). Las precondiciones deben ser parte de la interfaz de la rutina. Para el ejemplo anterior:

```
   get_parte( valor: INTEGER): REAL is
   require
         valor > 0
   do
        Result := get_total / valor
  ensure
         Result = get_total / valor
  end
```

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

En este, caso el responsable de verificar que el valor del parámetro sea mayor a cero es el cliente de la rutina. La precondición forma parte de la interfaz y por lo tanto es visible al cliente. El siguiente es un cuadro de los contratos de software considerando las pre y poscondiciones.

	Obligaciones	Beneficios
Proveedor	Satisfacer la poscondición	No debe preocuparse por las condiciones de la precondición
Cliente	Satisfacer la precondición	Obtiene el resultado establecido en la poscondición

Se puede entonces definir el principio de no-redundancia mencionado anteriormente: *Bajo ninguna circunstancia el cuerpo de una rutina deberá chequear el cumplimiento de la precondición.* La precondición compromete al cliente, ya que define las condiciones por las cuales una llamada a la rutina es válida. Las poscondiciones comprometen a la clase (donde se implementa la rutina) ya que establecen las obligaciones de la rutina. Es muy importante notar que la precondición y la poscondición que definen el contrato forman parte del elemento de software en sí.

ASERCIONES. PARTE 2

Es posible establecer aserciones más fuertes que otras. Ahora bien ¿qué significa que una aserción es más fuerte que otra? Se puede definir, que dadas dos aserciones P y Q, *P es más fuerte o igual que Q si P implica Q.* El concepto de fortificar o debilitar aserciones es usado en la herencia cuando es necesario redefinir rutinas. En Eiffel (y en general en las herramientas disponibles para otros lenguajes) el lenguaje para soportar aserciones tiene algunas diferencias con el cálculo de predicados: no tiene cuantificadores (aunque el concepto de agentes provee un mecanismo para especificarlos) , soporta llamadas a funciones y además existe la palabra reservada **old** (usada en las poscondiciones) para indicar el valor anterior a la ejecución de la rutina de algún elemento. Supóngase una clase que representa una cuenta bancaria que cuenta con una rutina *depositar* que recibe un importe como parámetro y agrega ese importe al saldo:

```
depositar (importe: REAL) is
    -- Depositar sum en la cuenta
    require
        cantidad_valida: importe >= 0
    do
          agregar_deposito(importe)
    ensure
          incremeto_balance: balance = old balance + sum importe
    end
```

La expresión old balance en este caso, indica el valor anterior a la ejecución de la rutina del balance. Es importante ver la diferencia existente entre la sentencia

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

agregar_deposito(sum)

y la expresión

balance = **old** balance + sum

El primer caso es prescriptivo y operacional, mientras el segundo es descriptivo y denotacional. Uno representa el *cómo* y otro representa el *qué*.

Como consecuencia directa de que las precondiciones forman parte de la interfaz de una rutina, surge la siguiente regla: *Toda función que aparezca en la precondición de una rutina r debe estar disponible a todo cliente al cual esté disponible dicha rutina r.* Si esto no fuera así, podría ser imposible para el cliente garantizar que se cumple la precondición antes de llamar a la rutina.

INVARIANTES DE CLASE

Los invariantes de clase sirven para expresar propiedades globales de las instancias de una clase, mientras que las pre y poscondiciones describen las propiedades de rutinas particulares. Desde el punto de vista de la metáfora de los contratos, los invariantes de clase establecen regulaciones generales aplicables a todos los contratos. Los invariantes de clase provienen del concepto de invariante de datos de Hoare (Hoare, 1972).

Por ejemplo la clase ARRAY en Eiffel cuenta (entre otros) con el siguiente invariante de clase:

```
invariant
   non_negative_count: count >= 0
```

Este invariante establece que un ARRAY no puede tener una cantidad de elementos menor que cero. Similarmente si se tiene una clase PERSONA se puede establecer como invariante que la edad debe ser mayor que cero y además que si es casada entonces el cónyuge tiene como cónyuge a sí mismo. En Eiffel se vería de la siguiente forma:

```
invariant
  edad_no_negativa:  edad >= 0
  matrimonio _ correcto:  soltero or else conyuge.conyuge = Current
```

En el invariante con etiqueta matrimonio _ correcto se establece que, o bien se es soltero o sino el cónyuge tiene como cónyuge a *Current* (o sea a sí mismo).

Los invariantes de clase deben satisfacerse en dos momentos: luego de la creación del objeto y luego de la llamada a una rutina por un cliente.

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

CORRECCIÓN DE UNA CLASE

Ahora se pueden establecer las condiciones de corrección de una clase. Para ello se usará la notación de la *tripleta de Hoare*. Una clase es correcta con respecto a sus aserciones sí y sólo si se cumplen las siguientes condiciones (Meyer,1997):

- Para toda rutina de creación ***rc***: {pre_{rc} } do_{rc} { $post_{rc}$ and INV}
- Para toda rutina exportada ***r***: {INV and pre_r } do_r { $post_r$ and INV}

Donde $pre_{a\ y}$ $post_a$ representan la pre y poscondición respectivamente de una rutina **a**, INV es el invariante de la clase donde se implementa **a** y do_a representa la ejecución de **a**.

De aquí se deduce que el rol principal del procedimiento de creación (algunas veces llamado constructor) es establecer el invariante de clase.

CONTRATOS Y AFIRMACIÓN DE CALIDAD.

¿Qué significa la violación de un contrato?. La violación de un contrato representa un defecto en el software. Dependiendo del tipo de aserción es la ubicación del defecto. Si se viola una precondición, entonces existe un defecto en el cliente. Si se viola la poscondición o el invariante de clase, existe un defecto en el proveedor. Es importante que se puedan monitorear las aserciones en tiempo de ejecución con propósitos de testeo, afirmación de calidad (las actividades de QA se basan en lo previsto por los contratos) y depuración. Es decir, no son solamente un mecanismo de especificación y documentación (en Eiffel existe una forma de visualizar una clase llamada *contract form* que muestra la clase con sus rutinas publicadas con pre y poscondiciones y el invariante de clase), sino que además cumplen un rol importante durante todas las etapas del desarrollo de software.

Es importante que el desarrollador tenga la opción de elegir en tiempo de compilación que aserciones monitorear. Luego de la depuración y el testeo, es posible generar el sistema de producción con las aserciones sin habilitar (evitando el chequeo en tiempo de ejecución). En general es recomendable mantener habilitado el chequeo de las precondiciones. Meyer hace una detallada explicación de esto (Meyer,1997). La violación de una aserción en tiempo de ejecución produce un excepción que puede ser posteriormente manejada.

Los contratos de software sirven como vehículo de comunicación entre desarrolladores, entre gerentes y desarrolladores, para realizar manuales, etc. Pero además son una de las herramientas más importantes para realizar reuso de software. El contrato permite conocer qué se está reusando y bajo qué condiciones. Un interesantísimo artículo de Jézéquel y Meyer dan cuenta de como la utilización de contratos podría haber evitado pérdidas millonarias en un proyecto de software (Jézéquel, 1997).

CONTRATOS Y HERENCIA

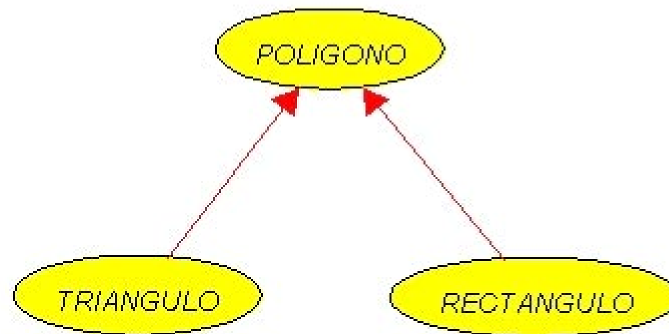
En este apartado, se discutirá como afecta la herencia a los contratos. Al generar una subclase de una clase existente, es posible que sea necesario redefinir algunas características. Surgen naturalmente las preguntas ¿qué pasa con los contratos? ¿qué ocurre con el invariante de clase?

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

Se analizará primero el caso del invariante de clase. Se rige por la *regla de la herencia del invariante* (*Invariant inheritance rule*) (Meyer,1997):

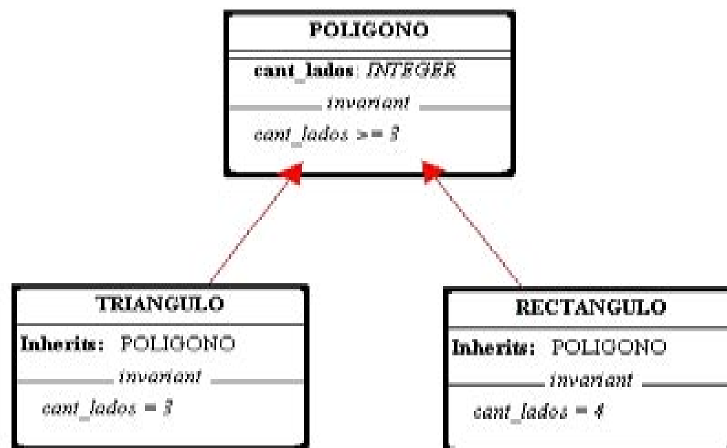
- La propiedad invariante de una clase es la conjunción lógica (*and*) de las aserciones que aparecen en su cláusula invariante y de las propiedades invariantes de sus padres, si hay alguno.

Se puede ver que esta regla es recursiva comprendiendo así a los invariantes de todos los ancestros. La regla anterior surge naturalmente de la propia naturaleza de la herencia. Las instancias de una clase son también instancias de los ancestros. Se tiene la siguiente jerarquía de clases:



Los triángulos y los rectángulos son a su vez polígonos. Los polígonos tienen por lo menos tres lados. Esta restricción se puede expresar como un invariante de clase : *cant_lados >= 3.* Para que los triángulos y rectángulos sean polígonos, deben respetar dicha restricción. Dada la regla de la herencia del invariante, ambos tienen como propiedad invariante *cant_lados >= 3.*

Ahora bien, como los triángulos tienen tres lados y los rectángulos cuatro lados, se puede agregar a cada una de estas clases, una cláusula invariante para expresar dichas restricciones.



La regla de herencia del invariante permite razonar acerca de la corrección de las clases y de la jerarquía de herencia. Si hubiera una subclase de polígono con una cláusula invariante indicando, por ejemplo, *cant_lados = 2*, dicha clase sería inválida ya que la propiedad invariante sería:

cant_lados >= 3 **and** *cant_lados = 2*

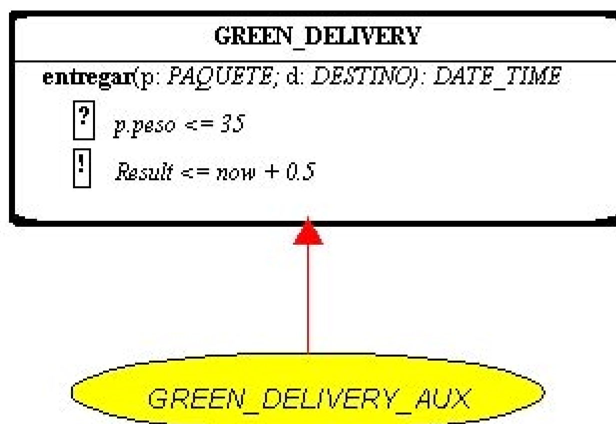
Esto indica que se está usando mal la herencia ya que, o bien, los polígonos pueden tener menos de tres lados o la nueva clase no es subclase de polígono (no es un polígono).

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

Cuando se redeclaran rutinas, es necesario respetar las aserciones de la rutina original. Es decir, no debería romperse el contrato establecido si se utiliza una rutina de una subclase. Las clases heredan los contratos de sus ancestros y deben respetarlos. Es posible, sin embargo, redefinir el contrato usando la metáfora del *subcontrato*. No siempre el que acuerda un contrato es el que lo lleva a cabo, en algunas ocasiones es posible subcontratar. Manteniendo las restricciones de tipificación, es posible que en tiempo de ejecución se ejecute una versión redefinida (vía herencia) de la rutina que tenía el contrato original; en dicho caso, la nueva rutina debe respetar el contrato. En el ejemplo de "*Green Delivery*", se tendría una clase llamada GREEN_DELIVERY con una rutina entregar que devuelve la fecha y hora de la entrega.

```
class GREEN_DELIVERY
feature
  entregar(p: PAQUETE, d: DESTINO): DATE_TIME is
        require
                peso_correcto:  p.peso  <= 35
        ensure
                entrega_a_tiempo: Result <= now + 0.5
```

Para simplificar, se muestra sólo la precondición correspondiente al peso del paquete y se asume una característica *now* que devuelve la hora actual. Supóngase ahora una subclase de GREEN_DELIVERY llamada GREEN_DELIVERY_AUX . En la figura se ve la situación: el símbolo ? representa una precondición y el símbolo ! la poscondición, de acuerdo a la notación BON (Waldén,1995)



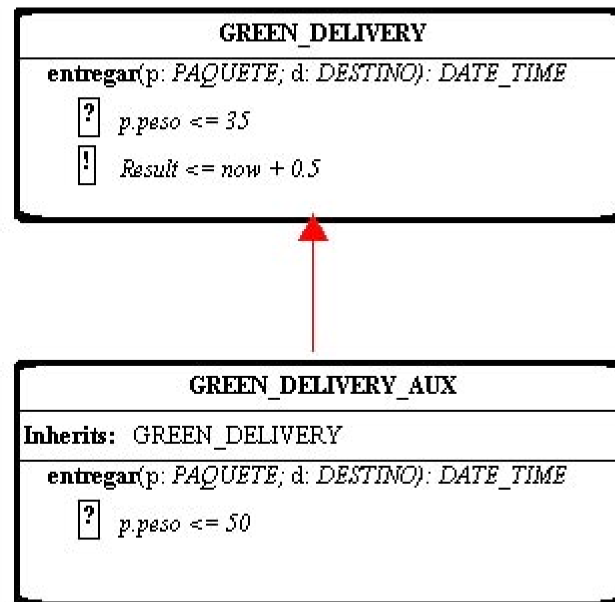
Supóngase que se redeclara la rutina *entregar* pero cambiando la precondición a paquetes de menos de 10 kilogramos. ¿Es esto correcto?. La redeclaración mencionada podría dar lugar a errores. Al tener enlace (*binding)* dinámico y comportamiento polimórfico, es posible que una variable declarada en el texto del programa como GREEN_DELIVERY, tenga asignado en tiempo de ejecución, un objeto instancia de GREEN_DELIVERY_AUX.

```
gd: GREEN_DELIVERY, gda: GREEN_DELIVERY_AUX
p: PAQUETE, d: DESTINO, f: DATE _ TIME
........ creación de objetos
.......  operaciones varias
......................
gd := gda – asignación polimórfica
.....................
f := gd.entregar( p, d)  -- ¿qué ocurre aquí?
```

Si el paquete pesa más de 10 kilogramos se tendría (a priori) una violación de la precondición ya que el tipo dinámico de *gd* es GREEN_DELIVERY_AUX. Pero, como fue declarado de tipo GREEN_DELIVERY, era razonable suponer que el paquete a enviar podía pesar hasta 35 kilogramos. Si la precondición de

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

GREEN_DELIVERY_AUX fuera que el peso del paquete puede ser hasta 50 Kg., no habría problemas ya que es una precondición más débil y por lo tanto las llamadas a la rutina *enviar* que cumplan con la precondición de GREEN_DELIVERY, también cumplirán con la precondición de GREEN_DELIVERY_AUX. La regla, en lo que respecta a precondiciones, para redefinir rutinas es sencilla:



- La nueva rutina debe mantener o debilitar la precondición de la rutina original

Dada la complejidad que implica para un compilador determinar si una aserción es más fuerte o débil que otra, lo que se hace en realidad es un *o lógico* entre la precondición de la rutina original y la nueva precondición de la rutina redeclarada. En Eiffel se escribe de la siguiente manera:

require else *nueva_precondición*

lo cual es lógicamente equivalente a: *precondición_original o sino nueva_precondición.*
En el ejemplo, la nueva precondición se leería como:

p.peso<=35 or p.peso<=50

El caso de las poscondiciones es diferente. Es importante que un subcontratista no exija más que lo establecido en el contrato original, pero no importaría que diera un servicio mayor. En el ejemplo de la entrega, la nueva subclase podría realizar la entrega en menos tiempo lo cual no perjudica, pero si sería perjudicial que la realizara en más tiempo. Esto lleva a la regla para redefinir poscondiciones:

- La nueva rutina debe mantener la poscondición original o hacerla más fuerte

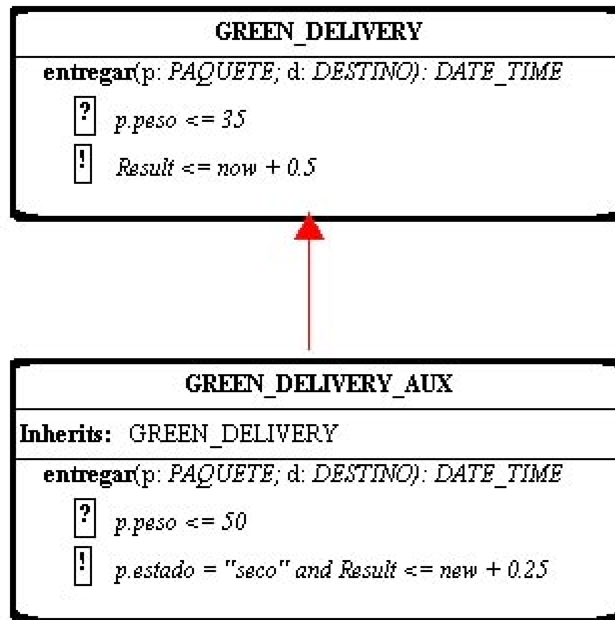
Ello en el lenguaje Eiffel se escribe:

ensure then *nueva_postcondición*

lo cual es lógicamente equivalente a: *poscondición_original y nueva_poscondición*

Por ejemplo, es posible en la rutina enviar de la clase GREEN_DELIVERY_AUX, redefinir la poscondición como se ve en la figura:

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.



En la nueva poscondición se asegura una entrega en un cuarto de hora y además que el paquete llega seco. Se leería de la siguiente manera:

(Result<= now + 0.5) and (p.estado="seco" and Result<=now +0.25)

La semántica de *require then y ensure else* evita que se puedan escribir precondiciones más fuertes y poscondiciones más débiles.

Las reglas que rigen los contratos en la herencia, posibilitan la construcción de clases abstractas con fuertes restricciones semánticas para sus subclases. Si se está construyendo una clase que posteriormente puede tener subclases, algunas veces es necesario garantizar las poscondiciones mediante un mecanismo que se llama *guarda.* Al relajar las precondiciones, puede llegarse a un estado en el cual las poscondiciones son difíciles o imposibles de cumplir. En el caso anterior, si la clase GREEN_DELIVERY_AUX redefine la rutina *entregar*, permitiendo paquetes de hasta 50 kilogramos pero se establece que los paquetes que pesan más de 35 Kilogramos, pueden entregarse con hasta una hora de demora, la poscondición se escribiría así:

ensure then Result <= new + 1

que indicaría *Result <= new + 0.5* ***and*** *Result <= new + 1.* Esto nunca sería posible para paquetes que demoren más de media hora. El mecanismo de guarda implica que al diseñar la clase original se preserven las poscondiciones para futuras subclases. Las poscondiciones preservadas (guarded) se escriben según el siguiente esquema:

(**old** precondición) *implies* postcondición

La palabra ***implies*** (implica) está indicando que cuando la parte izquierda de la aserción (old precondición) es satisfecha entonces la parte derecha es verdadera. En caso de que la parte izquierda no sea satisfecha no importa lo que pase con la parte derecha (equivalente a la implicación de la lógica de predicados). Para el ejemplo, una poscondición preservada sería escrita en la clase GREEN_DELIVERY como sigue:

ensure
(**old** p.peso <= 35) **implies** Result <= new + 0.5

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

De esta forma, al redefinir la poscondición como

ensure then Result <= new + 1

se está indicando:

((**old** p.peso <= 35) **implies** Result <= new + 0.5) **and then** (Result <= new + 1)

La nueva poscondición es justamente lo que se quería expresar: para los paquetes menores a 35 kilogramos se mantiene el contrato y para los mayores (que no estaban en el contrato original) se establece un tiempo de entrega de hasta una hora.

Para casos excepcionales pueden utilizarse precondiciones abstractas sin especificar completamente la aserción. (Meyer,1997)

PRINCIPIOS

En (Mitchel, 2002) R.Mitchell y J.McLim se enumeran una serie de principios a seguir cuando se diseñan clases utilizando contratos de software. Ellos realizan una extensa discusión y ejemplificación de dichos principios. En este artículo se enumeran los mismos:

Principio 1: *Separar consultas de comandos.* Este principio también es explicado detalladamente en (Meyer,1997). La idea es que las rutinas de una clase deben ser (en lo posible) o comandos o consultas pero no ambas cosas. Las consultas devuelven un valor (ej. funciones) y los comandos pueden cambiar el estado interno del objeto.

Principio 2: *Separar consultas básicas de consultas derivadas.* La intención es conseguir un conjunto de especificación formado por consultas que denominamos básicas, de tal forma que el resto de las consultas puedan derivarse de las básicas.

Principio 3: *Para cada consulta derivada escribir una poscondición especificando su resultado en términos de una o más consultas básicas.* Esto permite conocer el valor de las consultas derivadas conociendo el valor de las consultas básicas. Idealmente, sólo el conjunto minimal de especificación tiene la obligación de ser exportado públicamente.

Principio 4: *Para cada comando escribir una precondición que especifique el valor de cada consulta básica.* Dado que el resultado de todas las consultas puede visualizarse a través de las consultas básicas, con este principio se garantiza el total conocimiento de los efectos visibles de cada comando.

Principio 5: *Para toda consulta o comando decidir una precondición adecuada.* Este principio se auto explica ya que permite definir claramente el contrato de cada rutina.

Principio 6: *Escribir el invariante para definir propiedades de los objetos.* La idea aquí es ayudar al lector a construir un modelo conceptual apropiado de la clase.

Junto a estos principios los autores también explican una serie de guías y clases auxiliares para escribir contratos. Los principios enumerados junto al estudio de bibliotecas de clases (ej. EiffelBase) son una importante fuente de aprendizaje para el diseño de contratos.

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

CONTRATOS EN OTROS LENGUAJES

Si bien el lenguaje Eiffel tiene soporte nativo para diseño por contratos y el mismo es parte integrante de la cultura asociada al lenguaje, existen implementaciones para el soporte por contratos en otros lenguajes. Una de las más notorias es la implementación de contratos para el lenguaje JAVA llamada *iContract* (Kramer,1998). Básicamente *iContract* es un pre-procesador de código fuente que permite instrumentar invariantes de clase, pre y poscondiciones que pueden asociarse a métodos e interfaces en JAVA. Para lograrlo, utilizan marcas de comentario especiales como @pre y @post que al ser interpretadas por *iContract* se convierten en código para el chequeo de aserciones que se insertan en el código fuente. Las expresiones permitidas son un subconjunto del OCL (Object Constraint Language) de UML. Entre las características notables se incluye el soporte para cuantificadores (forall y exists). Cuenta además con soporte para la propagación de invariantes, pre y poscondiciones por medio de la herencia y la implementación de múltiples interfaces. Dado que los comentarios no son obligatorios, el código fuente que contenga anotaciones de diseño por contratos puede ser a su vez procesado por cualquier compilador Java que ignorará los mismos.

Entre las ventajas de *iContract* está el hecho de estar libremente disponible. Puede ser descargado desde http://www.reliable-systems.com/ contando además con soporte para *VisualAge for Java*. El siguiente fragmento de código muestra el uso de *iContract*:

```
class Person {

    protected age_;

     /**
      * @post return > 0
    */
     int getAge() {...}

     /**
        * @pre age > 0
     */
     void setAge( int age ){…}
     …
}
```

Hay otras implementaciones disponibles para JAVA. Entre ellas una denominada ***Jass***, por ***J**ava with* ***ass**ertions*, que también está basada en un pre-procesador. El siguiente código muestra un ejemplo:

```
public boolean contains(Object o) {
        /** require o != null; **/
        for (int i = 0; i < buffer.length; i++)
        /** invariant 0 <= i && i <= buffer.length; **/
        /** variant buffer.length - i **/
                if (buffer[i].equals(o)) return true;
        return false;
        /** ensure changeonly{}; **/
}
```

Hay una implementación de contratos de software realizada para los lenguajes *C#* y *VB.NET* en el marco de Microsoft .NET, llamada el *Design by Contract Framework*. Básicamente se trata de una librería que

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

provee un conjunto de métodos estáticos para definir precondiciones, poscondiciones e invariantes de clase junto a aserciones más generales. Se creó para ello un espacio de nombres (namespace) denominado DesignByContract. La sintaxis para las aserciones en C# es la siguiente:

```
Check.Require( [expresión boolena] );  //precondición

Check.Ensure( [expresión boolena]   );    //poscondición

Check.Invariant( [expresión boolena] );  // Invariante de clase
```

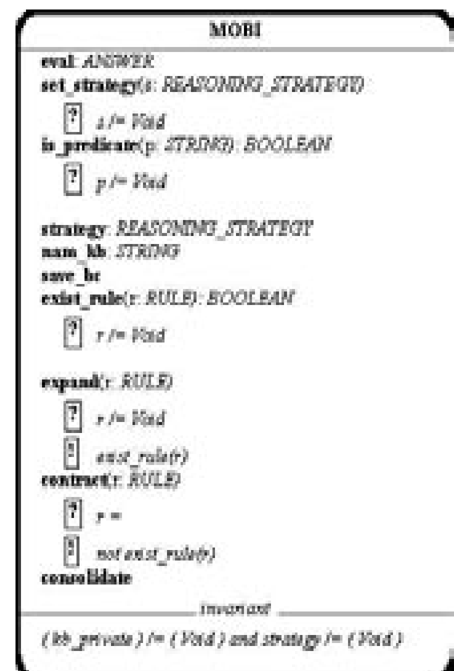
Esta implementación, que fue creada por Kevin McFarlane, también está libremente disponible en: *http://www.codeproject.com/csharp/designbycontract.asp*. Existen, además, implementaciones de diseño por contratos para C++, LISP y otros lenguajes (Plosch,1999) (Hunt,2000).

CONCLUSIONES Y REFERENCIAS

Se han presentado las ideas básicas del diseño por contratos y su aplicación en la construcción de software confiable. Las ideas vertidas, que tienen su origen en los métodos formales y que fueron desarrolladas por Bertrand Meyer, son la base de un curso de grado que los autores han dictado en la Universidad de Buenos Aires con la Prof. Graciela Matich[1]. Además soportan las investigaciones, que ellos están llevando a cabo, orientadas a la construcción de un marco de trabajo para agentes inteligentes (Rossel, 2003a). El marco de trabajo para agentes inteligentes llamado E-MOBI implementado en el lenguaje Eiffel, y que se diseña siguiendo los principios diseño por contratos, permite incorporar conocimiento basado en reglas a clases y objetos. Las características principales de E-MOBI son:

- Conocimiento basado en reglas
- Conocimiento privado para la instancias
- Herencia (múltiple) de conocimiento
- Operaciones de actualización dinámica de la base de conocimiento

A estas características se le agrega una extensión para soportar múltiples estrategias de razonamiento, que pueden ser cambiadas dinámicamente (Rossel, 2003b). La siguiente figura muestra el contrato resumido de la clase principal de E-MOBI:



Para más información sobre el lenguaje Eiffel, puede recurrirse al sitio http://www.eiffel.com y http://www.object-tool.com. Para una discusión de un método de desarrollo y notación llamado BON (Bussines Object Notation), que está fuertemente basado en las ideas de contratos de software y cuya referencia es el libro de Waldén y Jean-Marc (Waldén,1995), puede verse el sitio: http://www.bon-method.com/ . Las figuras de este artículo siguen dicha notación . Por último, un muy interesante artículo, relativo a diseño por contratos y componentes, es "*Making Components Contract Aware*" (Jézéquel,1999).

[1] Programación por Contratos.
http://www.cd.uba.ar/people/materias/ppc/

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.

BIBLIOGRAFÍA

DeMarco, Tom (1997) "Sine qua non of reuse". Letters. *IEEE* Computer, Vol. 30, No. 2, p 8.

Hoare, C.A.R. (1969) "An axiomatic Basis for Computer Programming", *Communications of ACM* Vol. 12, No. 10 pp 576-580

Hoare, C.A.R. (1972) "Proof of Correctness of Data Representations" *Acta Informática*, Vol. 1, No. 4, pp. 271-261.

Hunt, Andrew (2000) "Design by Contract in Ruby", *The Pragmatic Programmer*, LLC, August 2000

Jézéquel, Jean-Marc, Meyer, Bertrand (1997), "Design by Contract: The Lessons of Ariane" *IEEE Computer*, Vol. 30, No. 1 pp 129-130

Jézéquel, Jean-Marc, Train, Michel, Mingins, Christine (2000). *Design Patterns and Contracts. Addison* Wesley

Jézéquel, Plouzeau, Watkins,(1999) "Making Components Contract Aware" *IEEE Computer*, Vol. 32, No. 4. pp 38-45.

Kramer R, (1998) iContract - *The Java™ Design by Contract™ Tool* in Proceedings of Tools USA'98. IEEE Society Computer, Tools 26

Meyer, Bertrand (1992) "Applying 'Design by Contract'" *IEEE* Computer, Vol. 25, No. 10, pp 40-51.

Meyer, Bertrand (1992b). *Eiffel the Language Second Edition*. Prentice Hall.

Meyer, Bertrand (1997) *Object-Oriented Software Construction*, second edition. Prentice Hall

Mitchell, Richard, McKim, Jim (2002) *Design by Contract, by example*. Addison-Wesley.

Plosch, R, Pichler, J (1999) "Contracts: From Analysis to C++ Implementation", *Proceedings of the TOOLS-30 Conference*, August 1999, Santa Barbara.

Rossel, Gerardo, Manna Andrea (2003a) *E-MOBI Smart Object Model and Implementation*, programado para ser publicado en el *Journal of Object Technology*. Nov/Dec, 2003

Rossel, Gerardo, Manna Andrea (2003b) *Implementando múltiples estrategias de razonamiento en E-MOBI*. IV Workshop de Agentes y Sistemas Inteligentes. IX Congreso Argentino de Ciencias de la Computación: CACIC 2003. La Plata. Buenos Aires. Argentina.

Waldén Kim, Nerson Jean-Marc (1995) *Seamless Object-Oriented Software Architecture. Analysis and Design of Reliable Systems*. Prentice Hall

D.R. © Coordinación de Publicaciones Digitales. DGSCA-UNAM
Se autoriza la reproducción total o parcial de los artículos aquí presentados, siempre y cuando se cite la fuente completa y su dirección electrónica.