El Principio de Sustitución de Liskov

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

Dr. José Manuel Marqués Corral
*Licenciado en Matemáticas. Doctor en Informática. Profesor del Departamento de Informática de la **Universidad de Valladolid***
jmmc@.infor.uva.es

Con el estudio del principio de sustitución enunciado por Barbara Liskov en 1988, vamos a entrar en una de las facetas más controvertidas y potentes de la orientación a objetos, la herencia. Este principio intenta dictar unas reglas de diseño orientadas a la generación de unas jerarquías de clases basadas en un tipo muy concreto de herencia, la relación es_un (IsA)

En el número anterior se habló del principio abierto/cerrado, presentándolo como una de las bases fundamentales del Diseño Orientado a Objetos (DOO), siendo el encargado de fijar una filosofía de diseño de las clases, de forma que éstas puedan ser extendidas sin necesidad de modificar el código existente, obteniéndose así unas clases fáciles de mantener y reutilizar [1]. La abstracción y el polimorfismo son las características principales del paradigma objetual en las que se basa el principio abierto/cerrado. La herencia es el mecanismo de la orientación a objeto que mejor conjuga el uso de la abstracción y del polimorfismo, ya que ofrece la posibilidad de obtener clases derivadas que conformen las interfaces abstractas definidas por sus clases base abstractas.

La formulación de unas guías de diseño para la creación de jerarquías de herencia que no violen el principio abierto/cerrado, es el objetivo del principio de sustitución de Liskov, convirtiéndose así en una extensión lógica del principio abierto/cerrado.

Enunciado del principio de sustitución de Liskov

Barbara Liskov establece un principio en el que define una relación tipo/subtipo basada en el comportamiento, de forma que un tipo se considera subtipo de otro cuando cualquier instancia del primer tipo puede aparecer en cualquier lugar donde se espera una instancia del segundo. El enunciado de este principio es el siguiente: "*Si para cada objeto o_1 de tipo S hay un objeto o_2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o_1 es sustituido por o_2, entonces S es un subtipo de T*" [2].

El enunciado de este principio puede parecer un tanto abstracto, por lo que vamos a plantearlo desde un punto de vista más cercano a la programación orientada a objeto[1], con el fin de aclarar su significado. Así, este principio puede enunciarse como "*Las funciones que usan punteros o referencias a clases base deben ser capaces de utilizar objetos de las clases derivadas sin tener conocimiento de ello*", o lo que es lo mismo

[1] Tomaremos C++ como lenguaje orientado objetos para ilustrar los conceptos que vamos a ir presentando.

"*Las clases derivadas deben ser utilizables a través de la interfaz de la clase base, sin necesidad de que el usuario conozca la diferencia*".

La violación de este principio pone de manifiesto la importancia del mismo. Supóngase una función que utiliza un puntero a una clase base, pero que no cumple el principio de Liskov, entonces esta función debe conocer de forma explícita todas las clases derivadas de dicha clase base. Por tanto, esta función violaría el principio abierto/cerrado porque tendría que ser modificada cada vez que se tenga que crear una nueva clase derivada de la clase base.

Un ejemplo de violación del principio de sustitución de Liskov

Para ilustrar las consecuencias que puede conllevar la violación del principio de Liskov, vamos a suponer que existe una aplicación que funciona de forma correcta, pero además, dicha aplicación hace uso de la clase Rectángulo que aparece en la **Figura A**.

```
class Rectangulo {
        double alto, ancho;
public:
        Rectangulo(double _alto, double _ancho) {
                alto=_alto; ancho=_ancho;
        }
        void EstableceAlto(double tmp)   {alto=tmp;}
        void EstableceAncho(double tmp) {ancho=tmp;}
        double Alto()  const {return alto;}
        double Ancho() const {return ancho;}
};
```

Figura A. Clase Rectángulo.

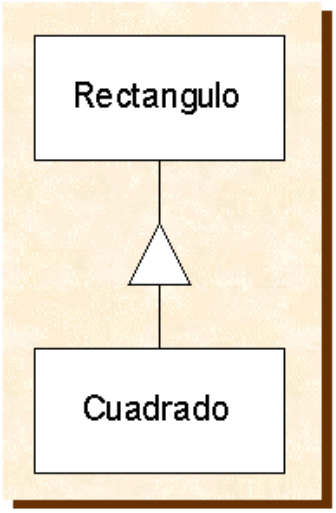

Figura B. Cuadrado hereda de Rectángulo

Al cabo de algún tiempo, los requisitos de los usuarios de dicha aplicación cambian, y requieren la posibilidad de trabajar con cuadrados. De esta forma, el diseñador de la aplicación decide modificar el diseño inicial, haciendo que la clase Cuadrado sea una clase derivada de la clase Rectángulo, tal y como se muestra en la **Figura B**.

Si se considera la relación de herencia como una relación **es_un** (*IsA*)[2], se tendría que un Cuadrado es_un Rectángulo, o lo que es lo mismo el tipo Cuadrado es un subtipo de Rectángulo. Según este diseño, y lo expresado por el principio de sustitución de Liskov, cualquier cliente que espere una instancia de la clase Rectángulo, debe ser capaz de trabajar con una instancia de la Clase Cuadrado sin tener que modificar el cliente, cumpliéndose así las máximas del principio abierto/cerrado.

[2] El uso de la relación IsA es una técnica ampliamente difundida en el Análisis y Diseño Orientado a Objetos [3], [4].

Pensar que un Cuadrado es un Rectángulo, es un hecho que puede parecer obvio a cualquier persona, y por tanto no es disparatado pensar que este diseño es adecuado y que cumple los principios abierto/cerrado y de Liskov. Pero, este diseño conlleva una serie de problemas bastante sutiles, que puede que no se aprecien hasta el momento de la implementación de la aplicación.

Para ver los problemas de este diseño, empecemos echando un vistazo a la clase Cuadrado que se muestra en la **Figura C**.

```
class Cuadrado: public Rectangulo {
public:
        Cuadrado(double lado):Rectangulo(lado, lado){}
        void EstableceAlto(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
        void EstableceAncho(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
};
```

Figura C. Clase Cuadrado.

La primera apreciación que puede hacerse es que la clase Cuadrado va a heredar los atributos de la clase Rectángulo (*alto, ancho*) aunque no necesita ambos, lo cual ya supone un desperdicio de memoria, que se hará más grave cuanto mayor sea el número de instancias de la clase Cuadrado. No obstante, el problema del desperdicio inútil de memoria podemos obviarlo, y así concentrarnos en otros problemas más interesantes.

La clase Cuadrado hereda también los métodos EstableceAlto() y EstableceAncho() de la clase Rectángulo, pero estos métodos violan la invariante de la clase Cuadrado, que establece que el alto y el ancho de un cuadrado deben ser iguales. Esto obliga a redefinir ambos métodos en la clase Cuadrado para sortear este problema de diseño, de forma que cuando cualquier cliente cambie el alto o el ancho de un objeto Cuadrado, se cambiará también la otra dimensión de forma automática, permaneciendo intacta la invariante del Cuadrado.

Teniendo en cuenta las definiciones de la clase Rectángulo y de la clase Cuadrado (**Figura A** y **Figura B** respectivamente), el programa principal de la **Figura D** funcionará sin problemas.

```
void main (void) {
        Cuadrado c1(2);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

        c1.EstableceAlto(5);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida 5 5

        c1.EstableceAncho(10);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida 10 10
}
```

Figura D. La invariante de los cuadrados se mantiene.

Pero, comprobemos si se cumple el principio de Liskov. Para ello, supongamos que a un determinado cliente, que espera un puntero o una referencia a un objeto de tipo Rectángulo, se le pasa una instancia de la clase Cuadrado. Por ejemplo, en la **Figura E** se muestra una función que recibe una referencia a un objeto de tipo Rectángulo y le cambia la relación de aspecto entre la anchura y la altura, de forma que ajusta la anchura a la mitad de su altura.

```
void CambiaAspecto(Rectangulo &r) {
        r.EstableceAncho(r.Alto()*.5);
}
```

Figura E. Cliente que recibe una referencia a un objeto de tipo Rectángulo.

Si se tiene el programa principal de la **Figura F**, al llamar a la función CambiaAspecto() con un objeto del tipo Cuadrado, ésta no mantendrá las invariantes del cuadrado porque invoca al método EstableceAncho() de la clase Rectangulo, no cambiándose la altura del objeto de tipo Cuadrado. Esta es una clara violación del principio de Liskov.

```
void main (void) {
        Rectangulo r1(8, 7);
        cout << r1.Alto() << " " << r1.Ancho() << endl;
        //Salida: 8 7

        CambiaAspecto(r1);
        cout << r1.Alto() << " " << r1.Ancho() << endl;
        //Salida: 8 4

        Cuadrado c1(2);
        cout << c1.Alto() << " " << c1.Ancho() << endl;
        //Salida: 2 2

        CambiaAspecto(c1);
        cout << c1.Alto() << " " << c1.Ancho() << endl;
        //Salida: 2 1
}
```

Figura F. Pérdida de la invariante del Cuadrado con la función CambiaAspecto().

La violación del principio de sustitución de Liskov por parte de la función CambiaAspecto(), se debe a que el diseño de la clase Rectángulo no estaba preparado para tener clases derivadas, por tanto sus métodos no soportan polimorfismo en tiempo de ejecución, el cual es necesario para que se cumpla el principio de Liskov. Se recuerda al lector que por defecto C++ tiene ligadura estática, lográndose la ligadura dinámica sólo cuando se declaran los métodos como virtuales en la clase base. Este problema es fácil de solucionar, pero conlleva una modificación de la clase base, lo cual es síntoma de un fallo de diseño.

En la **Figura G** se muestra la definición adecuada de la clase Rectángulo y de la clase Cuadrado, de forma que los métodos de la clase Rectángulo se han declarado como virtuales.

```
class Rectangulo {
        double alto, ancho;
public:
        Rectangulo(double _alto, double _ancho) {
                alto=_alto; ancho=_ancho;
        }
        virtual void EstableceAlto (double tmp) {alto=tmp;}
        virtual void EstableceAncho(double tmp) {ancho=tmp;}
        double Alto()  const {return alto;}
        double Ancho() const {return ancho;}
};

class Cuadrado: public Rectangulo {
public:
        Cuadrado(double lado): Rectangulo(lado, lado){}
        virtual void EstableceAlto(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
        virtual void EstableceAncho(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
};
```

Figura G. Nueva definición de las clases Rectángulo y Cuadrado.

Con los cambios introducidos en la definición de la clases Rectángulo y Cuadrado, el programa principal que se muestra en la **Figura H** al invocar a la función CambiaAspecto() no vulnera la invariante de los objetos de tipo Cuadrado, ya que se invoca al método EstableceAncho() redefinido en la clase Cuadrado, cambiando así su altura y su anchura.

```
void main (void) {
        Cuadrado c1(2);
        cout << c1.Alto() << " " << c1.Ancho() << endl;
        //Salida: 2 2

        CambiaAspecto(c1);
        cout << c1.Alto() << " " << c1.Ancho() << endl;
        //Salida: 1 1
}
```

Figura H. La invariante de la clase Cuadrado se mantiene.

Con las modificaciones realizadas se ha conseguido tener un diseño que no vulnera la invariante de la clase derivada, en este caso la clase Cuadrado. Pero, se ha tenido que

pagar un precio, el cliente CambiaAspecto() funciona de forma inadecuada cuando se le pasa un objeto de la clase Cuadrado, y en consecuencia los objetos de la clase Cuadrado no pueden ser sustituidos por los objetos de la clase Rectángulo.

Se puede codificar la función CambiaAspecto() de forma que si detecta que recibe un objeto de clase Cuadrado provoque una excepción. Pero, esta solución crea una dependencia de la función CambiaAspecto() con la clase Cuadrado**,** además cada vez que se vaya a crear una nueva clase derivada de la clase Rectángulo debe añadirse una comprobación en todos los clientes de la clase Rectángulo que estén en conflicto, vulnerándose de esta forma el principio abierto/cerrado (*ver **Listado 1***).

```
#include <iostream.h>
#include <typeinfo.h>

class Rectangulo {
	double alto, ancho;
public:
	Rectangulo(double _alto, double _ancho) {alto=_alto; ancho=_ancho;}
	virtual void EstableceAlto(double tmp) {alto=tmp;}
	virtual void EstableceAncho(double tmp) {ancho=tmp;}
	double Alto() const {return alto;}
	double Ancho() const {return ancho;}
};

class Cuadrado: public Rectangulo {
public:
	Cuadrado(double lado):Rectangulo(lado, lado){}
	virtual void EstableceAlto(double lado) {
		Rectangulo::EstableceAlto(lado);
		Rectangulo::EstableceAncho(lado);
	}
	virtual void EstableceAncho(double lado) {
		Rectangulo::EstableceAlto(lado);
		Rectangulo::EstableceAncho(lado);
	}
};

void CambiaAspecto(Rectangulo &);

void CambiaAspecto(Rectangulo &r) {
	if (typeid(r) == typeid(Cuadrado))
		throw "\nNo se puede ajustar el aspecto de un cuadrado.";
	r.EstableceAncho(r.Alto()*.5);
}

void main (void) {
	try {
		Rectangulo r1(8, 7);
		cout << r1.Alto() << " " << r1.Ancho() << endl; //Salida: 8 7

		CambiaAspecto(r1);
		cout << r1.Alto() << " " << r1.Ancho() << endl; //Salida: 8 4

		Cuadrado c1(2);
		cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

		CambiaAspecto(c1);
		cout << c1.Alto() << " " << c1.Ancho() << endl; //Levanta una excepción
	}
	catch(char *msg) {
		cout << msg << endl;
	}
}
```

Listado 1. Solución que viola el principio abierto/cerrado en la función CambiaAspecto().

Como conclusión se puede decir que un cuadrado puede ser un rectángulo, pero desde luego un objeto de la clase Cuadrado no es un objeto de la clase Rectángulo, porque el

comportamiento de un objeto Cuadrado no es consistente con el comportamiento de un objeto Rectángulo.

El principio de sustitución de Liskov establece que en DOO la relación IsA se refiere al comportamiento. No al comportamiento privado intrínseco, sino al comportamiento externo público, que es el comportamiento del que dependen los clientes.

Así, para cumplir el principio de Liskov, y por tanto el principio abierto/cerrado, todas las clases derivadas deben cumplir el comportamiento que esperan los clientes de sus clases bases.

¿Está la herencia limitada a la relación IsA?

La herencia suele asociarse de forma unívoca a la relación conceptual IsA, como puede apreciarse en el método de Booch [3] o en UML [4] por ejemplo. No obstante, existen importantes voces de la comunidad de orientación a objetos que defienden el uso de la herencia para expresar algo más que la relación IsA. En este sentido, cabe destacar la taxonomía que realiza Bertrand Meyer sobre la herencia, en la que presenta doce tipos diferentes de herencia, de los cuales sólo uno de ellos se corresponde con la relación IsA [5].

Dentro de la taxonomía de Meyer (**Figura I**) se presentan tres grandes familias:

- **Herencia de modelado:** Refleja relaciones entre las abstracciones de un modelo.
- **Herencia software:** Expresa relaciones software, que no son obvias en el modelo.
- **Herencia con variación:** Sirve para describir una clase mediante sus diferencias con otras clases.

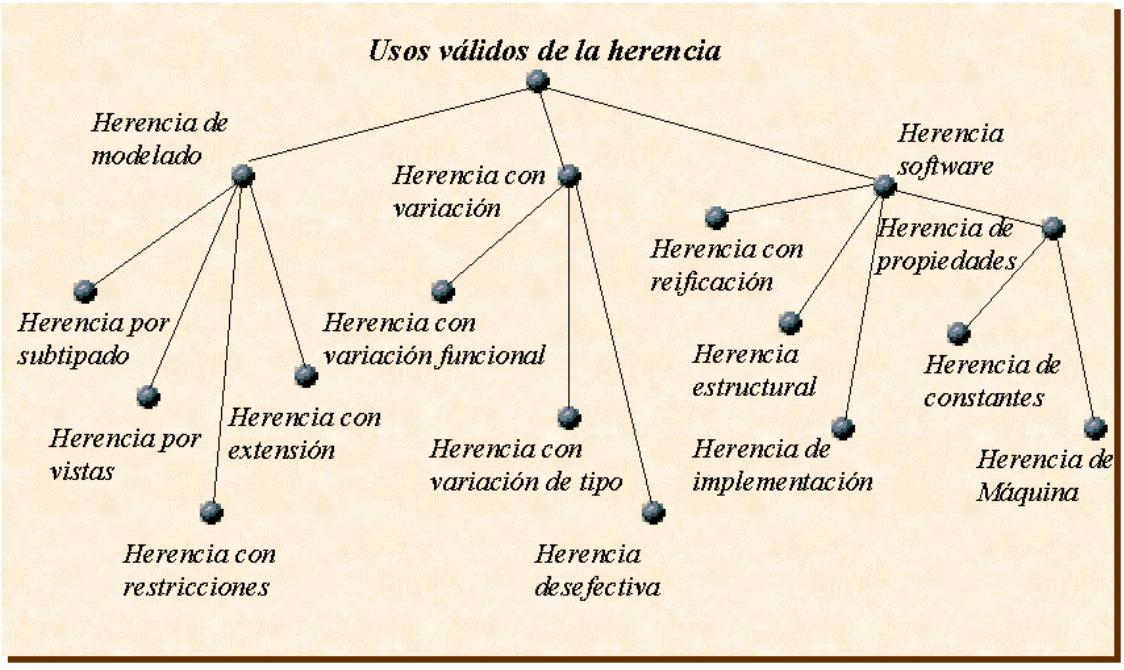

Figura I. Taxonomía de la herencia de Bertrand Meyer.

Conclusiones

Se ha presentado en este artículo otro de los principios fundamentales del DOO, el principio de sustitución de Liskov, muy relacionado con el principio abierto/cerrado. Las nociones de modelado que deben quedar claras después de este artículo son de nuevo la necesidad de contar con buenas abstracciones que sirvan de base para las jerarquías de herencia y para un uso correcto del polimorfismo en tiempo de ejecución. Volviendo al ejemplo del diseño expuesto anteriormente, el origen de todos los males se encuentra en no contar con una clase abstracta, por ejemplo Figura, de la cual pudieran heredar tanto la clase Rectángulo como la clase Cuadrado.

También cabe destacar la importancia que tiene la relación IsA en el modelado de jerarquías de herencia, y su repercusión en el comportamiento externo de los clientes. De todas formas, como se ilustra en la **Figura I**, no todo es IsA en el campo de la herencia.

Bibliografía

[1] **García Peñalvo, Francisco José y Pardo Aguilar, Carlos**. "*El Principio Abierto/Cerrado*". RPP, Nº39, Editorial América Ibérica. Abril, 1998.

[2] **Liskov, Barbara.** "*Data Abstraction and Hierarchy*". SIGPLAN Notices, Vol. 23(5), 1988.

[3] **Booch, Grady.** "*Análisis y Diseño Orientado a Objetos*". 2ª Ed. Addison-Wesley/Díaz de Santos, 1996.

[4] **Rational Software Corporation et al**. "*UML 1.1 Documentation Set*". http://www.rational.com/uml. 1 September 1997.

[5] **Meyer, Betrand.** "*Object-Oriented Software Construction*". 2nd Ed. Prentice-Hall, 1997.

[6] **García Peñalvo, Francisco José, Marqués Corral, José Manuel y Maudes Raedo, Jesús Manuel**. "*Análisis y Diseño Orientado al Objeto para Reutilización*". Technical Report (TR-GIRO-01-97V2.1), Universidad de Valladolid. Octubre 1997