Secretos de Herencia y Polimorfismo junto con Reflexión

Ricardo J. Alvarado Villalobos
Universidad de Costa Rica, Facultad de Ingeniería,
Escuela de Ciencias de la Computación e Informática, San Pedro, Costa Rica
rijoalvi@gmail.com
&
Jorge A. Loría López
Universidad de Costa Rica, Facultad de Ingeniería,
Escuela de Ciencias de la Computación e Informática, San Pedro, Costa Rica
kioto6@gmail.com

Abstract

Inheritance, polymorphism and reflection are relationships that belong to the high level programming languages. They've become really useful tools for programming all around the world. Each one of them has a lot of important characteristics for software creation, as it is method inheritance, overcharge, annulment, and many others that contribute to method reutilization and time and code improvement in a program.

Keywords: Inheritance, Polymorphism, Reflection, Methods Relationships, Programming Languages, Abstraction.

Resumen

La herencia, el polimorfismo y la reflexión son relaciones que existen en los lenguajes de alto nivel, que se han convertido en herramientas importantes para la programación en todo el mundo. Cada una de estas relaciones trae consigo muchas características ventajosas para la creación de software, como lo son la herencia múltiple, la herencia de métodos, la sobrecarga, la anulación, y otras que contribuyen a la reutilización de métodos, y a la mejora en cuanto a eficiencia en tiempo y cantidad de código de un programa.

Palabras clave: Herencia, Polimorfismo, Reflexión, Relaciones entre Métodos, Lenguajes de Programación, Abstracción.

1. Introducción

Desde que existen los algoritmos ha sido posible la programación, pero no fue hasta que se creó la computadora que esto se volvió tan relevante para las personas; desde los lenguajes de más bajo nivel, la programación ha ido evolucionando y cada vez crece más su nivel de abstracción. En este desarrollo, uno de los cambios más revolucionarios que ha habido ha sido la programación por objetos, que ha traído consigo nuevas formas de relacionar el código, como lo son la herencia, el polimorfismo y la reflexión. Más adelante veremos en detalle sus "secretos", y las características y ventajas que estas relaciones han traído a la programación

2. Clases, Objetos y relaciones entre objetos.

Para justificar la existencia de las relaciones entre objetos, debemos definir que es una clase y un objeto, para tener una idea de con que estamos tratando.

Con una clase, nos referimos a un conjunto de métodos y variables relacionados entre sí, que especifican la estructura de datos para cada instancia de la clase, y le dan operaciones para modificarla. Puede ser que algunos métodos puedan ser utilizados por otras clases, pero solamente la clase a la que pertenecen puede utilizar siempre la totalidad de sus métodos. Cuando se crea una instancia de una clase, construimos un objeto, del cual sus características y propiedades son definidas por la clase a la que pertenece, pero puede que no sea el único de la misma clase que exista. Los objetos pueden pedirle a otros objetos la ejecución de sus métodos, incluso pueden utilizar algunos métodos de otros, y así, al relacionar los objetos y las clases es que se crean programas.

2.1 Herencia

Las clases pueden heredar características de otras clases. Tomemos por ejemplo una clase estudiante_de_informatica. Un estudiante_de_informatica es una persona, por lo tanto come, respira, escucha y hace muchas otras cosas que hacen todas las personas. También tiene una cabeza, dos manos, dos ojos, etc., por lo que podemos decir que estudiante_de_informatica hereda de la clase persona, pues todos los "métodos", y la estructura de datos que tiene la clase persona, la tiene la clase estudiante_de_informatica, pero no viceversa, porque no todas las personas estudian informática, ni todas las personas trabajan con la computadora, entre muchas otras cosas.

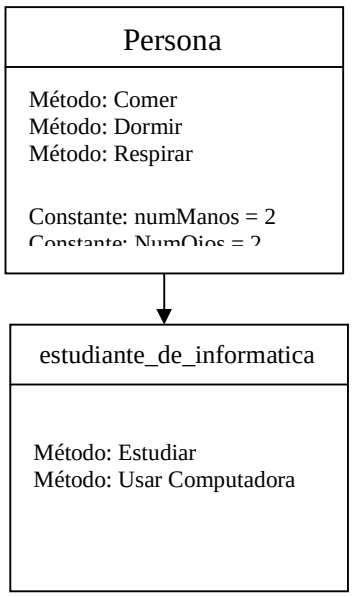


Fig. 1.1: Relación de herencia entre la clase estudiante y la clase estudiante_de_informatica.

Podemos definir entonces la herencia como “la creación de una clase especificando nada mas en que difiere de otra ya existente” **[3]**. Ésta siempre se va dar de forma jerárquica, teniendo como resultado que las clases superiores van a ser generalizaciones de las clases inferiores.

Desde el punto de vista de la subclase, podemos dividir la herencia en dos tipos: herencia simple, y herencia múltiple.

2.1.1 Herencia Simple

Como hemos venido analizando, una clase hereda de una clase paterna sus métodos y estructura de datos. Ahora, cuando solo hereda de una clase, es que se llama herencia simple. Puede ser que esta clase paterna herede de otra clase y que esto pase n veces, mientras que cada clase herede de una sola clase padre, estaremos ante herencia simple.

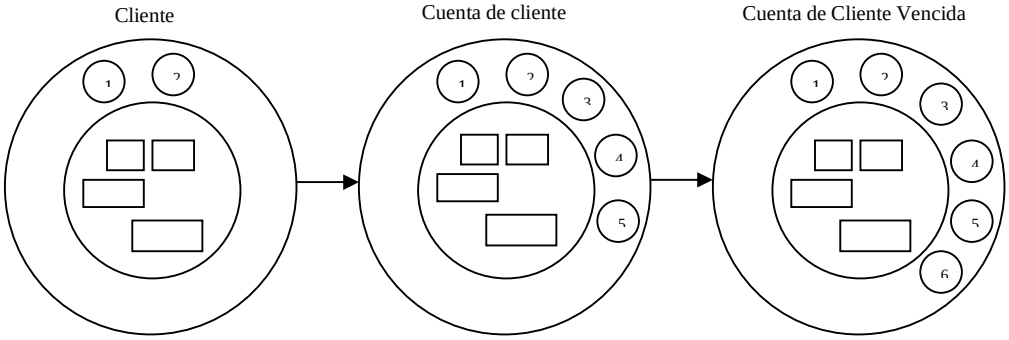


Fig. 1.2: Un ejemplo de Herencia Simple: Cuenta de Cliente Vencida hereda los métodos 1,2,3,4 y 5 de Cuenta de Cliente, que hereda a su vez los métodos 1 y 2 de Cuenta. [2]

2.1.2 Herencia Múltiple

En la herencia múltiple, una clase hereda estructuras de datos y operaciones de más de una clase padre. Este tipo de relación nos resuelve problemas que son imposibles con herencia simple: supongamos una clase vehículo anfibio: esta clase tiene características de un vehículo terrestre, y características de un vehículo acuático, por lo que hereda de ambas, y si vemos, estas clases son disjuntas, y no hay forma de mezclarlas, por lo que este tipo de problema sólo se puede resolver usando herencia múltiple.

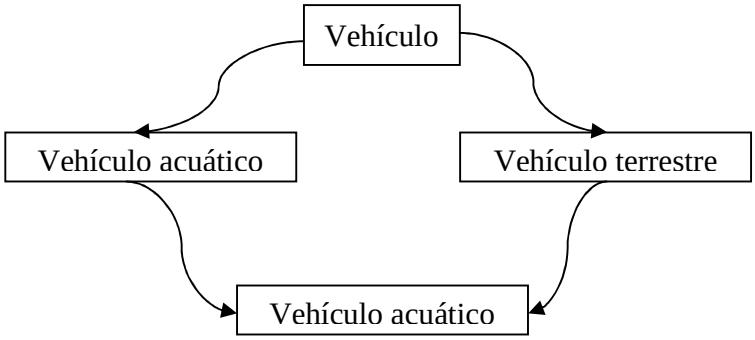
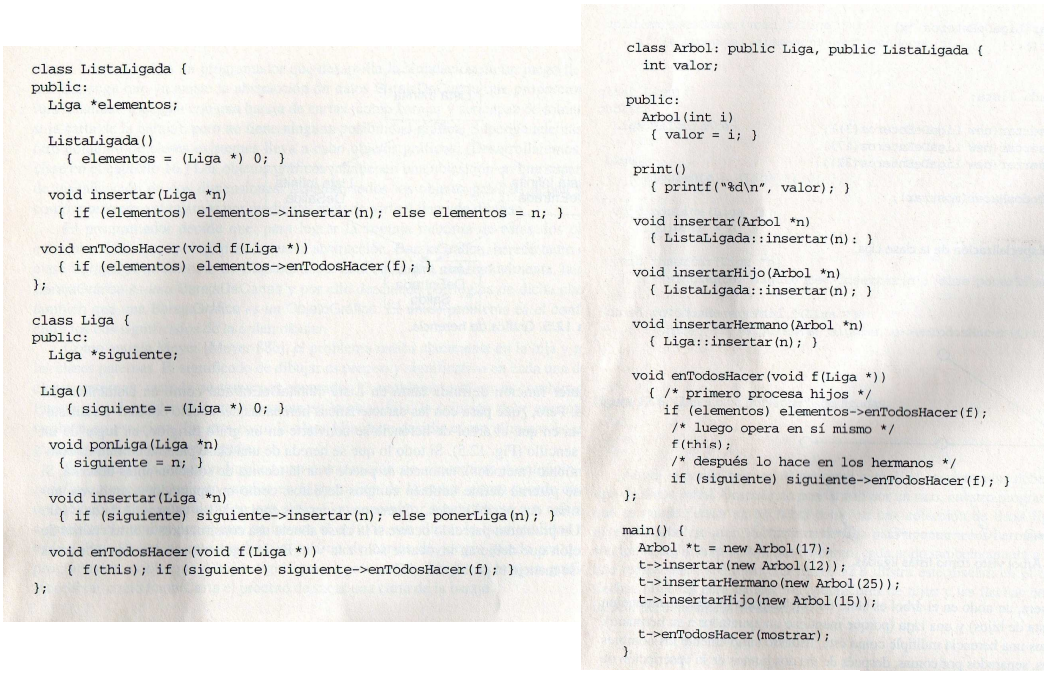


Fig. 1.3: Ejemplo de herencia múltiple: un vehículo anfibio hereda de un vehículo acuático y un vehículo terrestre. [4]

2.1.3 Herencia Múltiple en C++

Para ilustrar algunos de los conflictos que pueden aparecer al usar herencia, vamos a ejemplificar las formas de usarla en C++. Desgraciadamente al haber aparecido la herencia, también trajo consigo problemas, como los que veremos a continuación:



```
class ListaLigada {
public:
  Liga *elementos;

  ListaLigada()
    { elementos = (Liga *) 0; }

  void insertar(Liga *n)
   { if (elementos) elementos->insertar(n); else elementos = n; }

 void enTodosHacer(void f(Liga *))
   { if (elementos) elementos->enTodosHacer(f); }
};

class Liga {
public:
  Liga *siguiente;

 Liga()
    { siguiente = (Liga *) 0; }

  void ponLiga(Liga *n)
   { siguiente = n; }

  void insertar(Liga *n)
   { if (siguiente) siguiente->insertar(n); else ponerLiga(n); }

  void enTodosHacer(void f(Liga *))
    { f(this); if (siguiente) siguiente->enTodosHacer(f); }
};
```

```
class Arbol: public Liga, public ListaLigada {
  int valor;

public:
  Arbol(int i)
   { valor = i; }

 print()
   { printf("%d\n", valor); }

 void insertar(Arbol *n)
   { ListaLigada::insertar(n); }

 void insertarHijo(Arbol *n)
   { ListaLigada::insertar(n); }

 void insertarHermano(Arbol *n)
   { Liga::insertar(n); }

 void enTodosHacer(void f(Liga *))
   { /* primero procesa hijos */
     if (elementos) elementos->enTodosHacer(f);
      /* luego opera en sí mismo */
      f(this);
      /* después lo hace en los hermanos */
      if (siguiente) siguiente->enTodosHacer(f); }
};

main() {
  Arbol *t = new Arbol(17);
  t->insertar(new Arbol(12));
  t->insertarHermano(new Arbol(25));
  t->insertarHijo(new Arbol(15));

  t->enTodosHacer(mostrar);
}
```

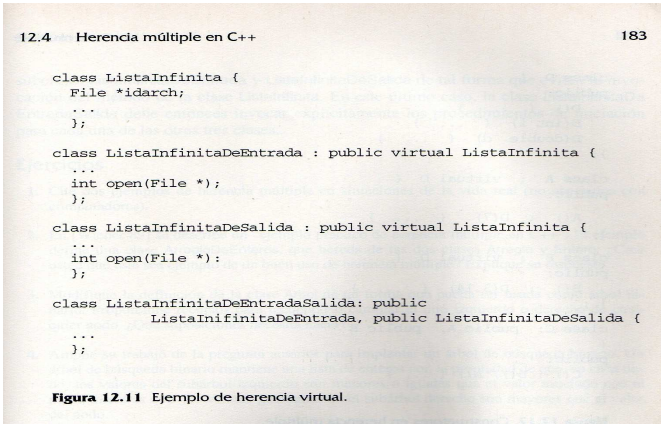
Fig. 1.4: Ejemplo de herencia múltiple en C++: la clase Arbol hereda de la clase Liga y de la clase ListaLigada [1]

Al ver este ejemplo podemos ver ciertas ambigüedades que se presentan al programar: en primera instancia, se debe manejar la operación insertar. En este código se puede insertar un nodo hijo, o se puede insertar un nodo hermano; la primera operación la da la operación insertar de la clase ListaLigada, la segunda la proporciona la clase Liga. En el ejemplo, el programador decidió que la operación insertar se referirá a insertar un hijo, pero también implementa dos nuevas funciones que especifiquen explícitamente el propósito, pero que son, en cierto sentido sólo son renombramientos, sin crear una nueva función.

Cuando vemos el método TodosHacer, apreciamos que puede haber una ambigüedad mayor incluso que la del ejemplo anterior, ya que aquí la acción apropiada sería ejecutar un recorrido del árbol, por ejemplo preoden. En este recorrido se visitan primero los nodos hijos, seguidos por el nodo actual, y finalmente los hermanos, que visitarán recursivamente a sus hijos. Así, la ejecución es una combinación de las acciones que proporcionan las clases Liga y ListaLigada.

2.1.4 Herencia Virtual en C++

Este tipo de herencia se utiliza cuando se quiere que una clase no herede de sus padres los campos de datos que éstos tienen. La palabra clave en este tipo de herencia es "virtual", que significa que la clase referida como virtual puede aparecer muchas veces en clases descendientes de la clase actual, pero que sólo deberá incluir una copia de la superclase. Para darnos un ejemplo más claro, la figura 1.5 nos muestra las declaraciones de cuatro clases ListaInfinita, ListaInfinitaDeEntrada, ListaInfinitaDeSalida y ListaInfinitaDeEntradaSalida, de esta última se quiere que sólo herede una copia de los campos de datos definidos en ListaInfinita, clase de la cual Heredan ListaInfinitaDeEntrada y ListaInfinitaDeSalida.



12.4 Herencia múltiple en C++ 183

```
class ListaInfinita {
  File *idarch;
  ...
  };

class ListaInfinitaDeEntrada : public virtual ListaInfinita {
  ...
  int open(File *);
  };

class ListaInfinitaDeSalida : public virtual ListaInfinita {
  ...
  int open(File *):
  };

class ListaInfinitaDeEntradaSalida: public
          ListaInifinitaDeEntrada, public ListaInfinitaDeSalida {
  ...
  };
```

Figura 12.11 Ejemplo de herencia virtual.

Fig. 1.5: Ejemplo de Herencia Virtual en C++ [1]

2.1.5 Ventajas de la Herencia

Las mayores ventajas que presenta la herencia son el ahorro de código y tiempo, ya que al heredar métodos y estructuras de datos es mucho más rápido que se puede crear un programa. En esto también es una ventaja que los objetos pueden ser construidos a partir de otros similares, y todo esto aumenta la eficiencia tanto del programador como del programa.

2.2 Polimorfismo

El polimorfismo, según la Real Academia Española, se define como la cualidad de lo que tiene o puede tener distintas formas [5]. En lenguajes de programación, un objeto polimórfico es una entidad a la que se le permite tener valores de tipos diferentes a través de la ejecución. Su fuerza radica en que permite que los algoritmos de alto nivel se escriban una vez, y se reutilicen en forma repetida con diferentes abstracciones de bajo nivel. Con esto, los algoritmos de alto nivel se convierten en herramientas que un programador puede reutilizar muchas veces sin volver a codificarlos al almacenarlos en bibliotecas.

Si nos referimos al polimorfismo en los lenguajes orientados a objetos, su fortaleza radica en que los objetos pueden combinarse en muchas formas diferentes, con lo que producen diversas técnicas para compartir y reutilizar código.

La forma más común del polimorfismo en los lenguajes de programación es la sobrecarga, aunque no es la única que existe. A continuación analizaremos esta y otras formas de polimorfismo.

2.2.1 Sobrecarga

Cuando una función está sobrecargada, ésta tiene dos o más cuerpos de funciones asociadas a su nombre. En este tipo de polimorfismo, el nombre de la función es el que es polimórfico. Otra forma de ver esto es pensar en una sola función abstracta que recibe muchos tipos diferentes de argumentos, y su código ejecutado depende del argumento que le den.

El ejemplo más común para definir la sobrecarga es suponer que tenemos una clase NumeroRacional, que puede ser tanto un número entero como un double. Ahora bien, supongamos que en C++ no está definida la suma entre un int y un double. Lo que podemos hacer es sobrecargar el operador '+' para que se puedan realizar operaciones de este tipo sin ningún problema. Veamos cómo se haría esto en C++:

```
Class NumeroRacional
{
    public:
        // Permite sumar un NumeroRacional con un double
        NumeroRacional operator + (double sumando);
};
```

Fig. 1.6: Ejemplo de sobrecarga del operador suma

De esta manera, queda utilizable el operador '+' para realizar sumas de entero+entero, entero+double, double+entero, y double+double.

2.2.2 Anulación

Suponga que en una clase está definido un método general definido para un mensaje particular. La mayoría de las subclases que heredan de ella también heredan y usan el mismo método, pero resulta que en una subclase se define un método con el mismo nombre, lo que provoca que este método oculte el acceso desde él al método general para instancias de esa clase, por lo que decimos que este método anula al primero.

La anulación suele ser transparente para el usuario de la clase, y como en la sobrecarga, muchas veces se piensa que las dos funciones son en realidad una sola (por ejemplo se piensa en la sobrecarga que la suma de enteros y de doubles es lo mismo).

2.2.3 Forma en que se implementa el polimorfismo, las funciones virtual y la vinculación dinámica

C++ hace que el polimorfismo sea más fácil de programar. Ciertamente es posible programar en base al polimorfismo en lenguajes no orientados a objetos como C, pero para ello se requiere de una compleja y potencialmente peligrosa manipulación de apuntadores. Hablaremos ahora de como C++ puede implementrar, internamente, el polimorfismo, las funciones virtuales y la vinculación dinámica.

El compilador de C++ cuando compila una clase que tiene una o más funciones virtuales, crea una *tabla de funciones virtuales(vtable)* para esa clase. Durante la ejecución de un programa, este utiliza la vtable para seleccionar la implementación de la función apropiada cada vez que se llama a una función virtual de esa clase. En cada entrada de la vtable de la clase se almacena un puntero a la función virtual si ésta cuenta con implementación, con un puntero hacia la implementación de la función de la clase padre, o bien un puntero nulo si la función es virtual pura y, por lo tanto, carece de la misma.

Además de la vtable el compilador genera, cada vez que se instancia un objeto de una clase con funciones virtuales, un apuntador a la vtable de esa clase. Y luego, también, un manejador de objeto que recibe la llamada a la función virtual. Este manejador también puede ser una referencia.

Cuando se realiza una llamada a una función virtual desde un puntero de la clase base, el compilador determina cual es el numero de la función en la vtable. Con esto genera el desplazamiento correspondiente en código máquina para la llamada a la función. Seguidamente va des-referenciando los apuntadores desde el apuntador de la clase base pasando por el apuntador del objeto de la clase derivada, y con este último puntero llega hasta la vtable. Suma el desplazamiento antes mencionado y ejecuta el código correspondiente a la función de la clase derivada.

Estas estructuras y el manejo de ellas parece complicado, pero es el compilador el que se encarga de generar todo el código que se encarga de ello, dejando así transparente al programador todo este manejo.

Por último se debe señalar que toda esta des-referenciación de apuntadores requiere tiempo adicional de ejecución. Además las vtables y demás ocuparan espacio adicional en memoria

.

2.3 Reflexión

Básicamente, la reflexión es la capacidad de un programa de revisarse y/o modificarse en tiempo de ejecución. Además de conocer los métodos de una clase, así como sus parámetros, nombres y valores de retorno. Está ligada a la programación orientada a objetos, pues facilita el uso de la misma y aprovecha las cualidades de este paradigma de programación. Igual se puede implementar en lenguajes que no sean orientados a objetos. Algunos ejemplos de lenguajes que soportan la reflexión son: 3LISP, 3KRS, ObjVLisp.

La reflexión nos permite evitar el uso de if para determinar que clase debe procesar cierta información.

2.3.1 Conceptos de reflexión

Para utilizarla se deben conocer dos conceptos de cómputo de ejecuciones que son:
-Cómputo atómico: se completa en un simple paso lógico.
-Cómputo compuesto: requiere sub-cómputos para terminar de ejecutarse.
Algunos ejemplos de cómputos atómicos son números, referencias a variables, y el manejo de operaciones primitivas. Algunos ejemplos de cómputos compuestos serían funciones, segmentos de condición, y operaciones secuenciales.

-Meta información: es donde se guarda la información de un objeto a un meta nivel. De esta manera se obtiene la información requerida por la reflexión. Por ejemplo para saber que tanto trabaja un proceso, esto se lograría contando las instrucciones que ejecuta el proceso. Pero hacer esto es muy engorroso mediante otro programa. Pero con la reflexión es sencillo porque se puede programar desde un meta nivel.

2.3.2 Principios y utilidad de la reflexión en Java.

La reflexión se puede utilizar en el lenguaje java porque partimos del hecho de que todos los objetos en java heredan de la clase *java.lang.Object*, y por lo tanto todos tienen un método *getClass*, cuya firma pública es *public final Class getClass()*.Este método nos devuelve un objeto *java.lang.Class*, que nos abre las puertas para trabajar con la reflexión.

Con eso, con sólo importar la clase *java.lang.reflect.** y utilizar sus diferentes métodos, podemos obtener el nombre de una clase, el nombre de sus variables públicas, el nombre de sus métodos públicos, el tipo del que es cada método, los parámetros que recibe cada uno, las excepciones que puede lanzar cada método, y mas que eso, podemos modificar los valores sus campos, no solo públicos, sino también los privados, si existen los métodos set y get para ese campo.

2.3.3 Ejemplo:

Este ejemplo en Java sirve para instanciar una clase sin saber su nombre:

```
//Primero se crea la interfaz.

public interface IInterface {
    public int operacion(int a, int b);
}

//Luego se crean las clases de servicio:

//CLASE A
public class A implements IInterface{
    private int s1, s2;

    public A() {
        s1 = 0;
        s2 = 0;
    }

    public A(int a, int b) {
        s1 = a;
        s2 = b;
    }

    public int operacion(int a, int b) {
        return a+b;
    }
}

//CLASE B
public class B implements IInterface{
```

```
    public int operacion (int a, int b) {
        return a-b;
    }
}

//Para obtener un objeto a partir de una clase, de la cual no se conoce el
//nombre:

String clase = "A";
Class c = Class.forName(clase);

//En este momento el objeto 'c' tiene la información referente a la clase A.
```

Fig. 1.7: Ejemplo de reflexión en Java.

2.3.4 Reflexión en C++: el RTTI

Hasta ahora hemos visto que la reflexión se puede utilizar muy fácilmente en el lenguaje java, pero, ¿es posible crear una forma de reflexión en C++?. Sabemos que en C++ las clases no heredan de una superclase como la *java.lang.Object*, pero trataremos de explicar una forma de lograr muchas partes de la reflexión mediante un sistema de C++ llamado RTTI.

RTTI significa "Run-Time Type Information", y se refiere a un sistema que guarda información del tipo de datos de un objeto en tiempo de ejecución. Esta información puede ser desde tipos de datos simples (como enteros o char) hasta objetos genéricos.

Para lograr esta reflexión, contamos con el método *dynamic_cast* de C++, el cual nos permite en tiempo de ejecución realizar una conversión del tipo de un objeto, y en caso de que no se pueda realizar, se obtiene un puntero nulo. También tenemos al método *typeid*, que devuelve una referencia a la clase standard *std::type_info*, que contiene información acerca del tipo de esa clase. Esta clase type_info tiene operadores interesantes, como los de comparación == y !=, aunque el más útil es el siguiente:

```
Const char* std::type_info::name() const;
```

Este método es el que nos va a permitir obtener realmente alguna cualidad de la reflexión, ya que podemos obtener de la siguiente forma una cadena de caracteres con el nombre verdadero de la clase del objeto:

```
const std::type_info &información = typeid (punteroObjetoDesconocido);
std::cout << información.name();
```

De esta forma, se puede lograr al menos algunas de las características de la reflexión en C++.

3. Conclusiones.

Después de esta investigación, nos queda claro que la herencia, el polimorfismo y la reflexión son tipos de relaciones que surgieron con gran fuerza con la aparición de los lenguajes de programación por objetos. También concluimos que los tres están relacionados de cierta forma, y se pueden utilizar a la vez en una misma clase.

Creemos que es muy importante en la programación estos tipos de relaciones, y que realmente contribuyen a la práctica y eficiente programación, y que con su nivel de abstracción contribuyen sobre todo al buen entendimiento del código, ya que son relaciones que las personas vemos de forma natural. El único secreto que existe en cuanto a estas ellas es la cantidad de usos que pueden dársele si se comprenden bien los conceptos, y se saben utilizar bien estas herramientas.

Finalmente, creemos que es importante conocer lo que se puede lograr con las características de cada lenguaje, como en el caso de los RTTI en C++, que hacen posible una implementación de algunas características de la reflexión, sin tener las herramientas que se dan en java para obtener ésta misma.

4. Referencias.

[1] Budd, Timothy.: Introducción a la Programación Orientada a Objetos. Editorial Addison-Wesley Iberoamericana. 1994.

[2] Martin, J., Odell, J.: Análisis y Diseño Orientado a Objetos. Editorial Prentice Hall Hispanoamericana, S.A. 1994.

[3] Siegel, Shel.: Object Oriented Software Testing: a Hierarchical Approach. Editorial Wiley Computer Publishing. 1996.

[4] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenser, W.: Modelado y Diseño Orientado a Objetos, Metodología OMT. Editorial Prentice Hall Internacional Ltd. 1991.

[5] Diccionario de la Real Academia Española, http://www.rae.es.

[6] Sobel, J., Friedman, D. An Introduction to Reflection-Oriented Programming. Indiana University, http://www.cs.indiana.edu/~jsobel/rop.html.

[7] Sun Microsystems, Java Reflection Tutorial, http://java.sun.com/docs/books/tutorial/reflect/index.html.

[8] Vollmann, D., Aspects of reflection in C++, Vollmann engineering gmbh, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1751.html.

[9] Chiu, Kenneth.; C++ Reflection. www.cs.binghamton.edu/~kchiu/cs580c-s05/slides/C++%20Reflection.ppt

[10] Msdn Lybrary, http://msdn.microsoft.com/es-es/library/ms384355(VS.71).aspx

[11] Jenkov, Jacob.; Java Reflection Tutorial, http://tutorials.jenkov.com/java-reflection/index.html

[12] Crafton, Jim.; Using the C++ RTTI/Reflection APIs in the VCF, http://www.codeproject.com/KB/library/vcf_rtti.aspx

[13] Knizhnik, Konstantin.; Reflection, http://www.garret.ru/cppreflection/docs/reflect.html#architecture

[14] Kalev, Danny.; Use RTTI for Dynamic Type Identification, http://www.devx.com/getHelpOn/Article/10202/1954