LOGALI

TEORÍA

Polimorfismo, asociación y composición

SAP ABAP Programación Orientada a Objetos

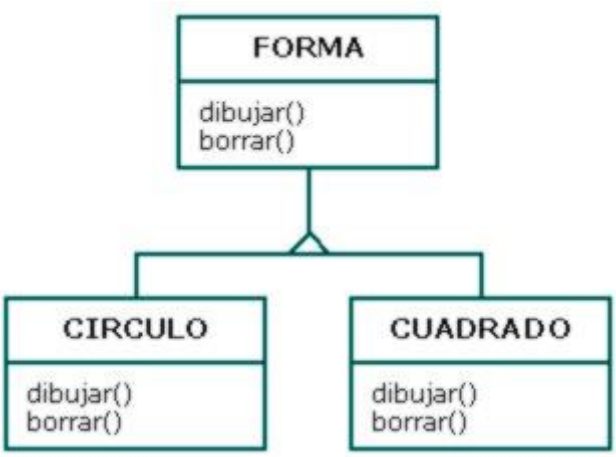
Contenido

1. Polimorfismo

1.1. Conceptos

Polimorfismo es la capacidad de un objeto de adquirir varias formas. El uso más común de polimorfismo en la programación orientada a objetos se da cuando se utiliza la referencia de una clase padre, para referirse al objeto de la clase hija. Esta característica permite definir distintos comportamientos para un método dependiendo de la clase sobre la que se realice la implementación. En todo momento tenemos un único medio de acceso, sin embargo se podrá acceder a métodos distintos. Es importante saber que la única manera de acceder a un objeto es a través de una variable de referencia. La variable de referencia sólo puede ser de un tipo. Una vez declarado el tipo de la variable de referencia, no se puede cambiar.

Veamos el siguiente ejemplo, en el que se define una clase FORMA de la que se heredan las clases CIRCULO y CUADRADO.



La clase forma define los métodos dibujar y borrar. En la definición de estos métodos se implementará el código común a todos los objetos de la clase. Sin embargo cuando definamos las clases hijas, círculo y cuadrado, será necesario modificar estos métodos para adaptarlos a las nuevas subclases. El método de la clase padre implementa aquellas acciones comunes. Las clases hijas añaden las operaciones particulares que necesiten. Cuando utilicemos los métodos de la clase forma no tendremos que hacer distinción entre cuadrados y círculos. Gracias al polimorfismo se ejecutará el método adecuado en función de la subclase a la que pertenezca el objeto.

1.2. Polimorfismo con clases

Para implementar el polimorfismo con clases debemos tener herencia entre las clases y que exista la redefinición de los métodos en las subclases.

Veamos el siguiente ejemplo. Tenemos la clase superior CL_ELEMENTO_GRAFICO y la subclase CL_LINEA_VERTICAL. En la clase superior tenemos implementado el método PINTA, método que se ha redefinido en la subclase.

```
CLASS cl_elemento_grafico DEFINITION.

  PUBLIC SECTION.
    METHODS pinta.

ENDCLASS.

CLASS cl_elemento_grafico IMPLEMENTATION.

  METHOD pinta.
    WRITE / 'Elemento grafico'.
  ENDMETHOD.

ENDCLASS.

CLASS cl_linea_vertical DEFINITION INHERITING FROM cl_elemento_grafico
.

  PUBLIC SECTION.
    METHODS pinta REDEFINITION.

ENDCLASS.

CLASS cl_linea_vertical IMPLEMENTATION.

  METHOD pinta.
    WRITE / 'Línea vertical'.
  ENDMETHOD.


ENDCLASS.
```

En este caso podemos utilizar una referencia de la clase superior para llamar al método **PINTA** y crear un programa genérico. Si la referencia apunta a un objeto de la clase hija se va llamar al método implementado en la subclase.

```
DATA: go_elemento_grafico TYPE REF TO cl_elemento_grafico,
      go_linea_vertical   TYPE REF TO cl_linea_vertical.

go_linea_vertical = NEW ( ).

go_elemento_grafico = go_linea_vertical.

go_elemento_grafico->pinta( ).
```

En cuando se llama al método PINTA sobre la referencia del padre se ejecuta la redefinición del método implementado en la subclase CL_LINEA_VERTICAL.

1.3. Polimorfismo con interfaces

Igual que con las clases se puede conseguir el polimorfismo con las interfaces, porque al implementar una interfaz podemos decir que el objeto de la clase es un objeto de la interfaz.

Veamos el siguiente ejemplo donde la clase CL_LINEA_HORIZONTAL implementa la interfaz IF_GRAFICO que tiene definido el método PINTA.

```
INTERFACE if_grafico.
  METHODS pinta.
ENDINTERFACE.

CLASS cl_linea_horizontal DEFINITION.

  PUBLIC SECTION.
    INTERFACES if_grafico.

ENDCLASS.

CLASS cl_linea_horizontal IMPLEMENTATION.

  METHOD if_grafico~pinta.
    WRITE / 'Línea horizontal'.
  ENDMETHOD.

ENDCLASS.
```

A la referencia de interfaz le asignamos una instancia de la clase y llamamos al método PINTA.

```
DATA: go_grafico          TYPE REF TO if_grafico,
      go_linea_horizontal TYPE REF TO cl_linea_horizontal.

go_linea_horizontal = NEW ( ).

go_grafico = go_linea_horizontal.

go_grafico->pinta( ).
```

Como la referencia de la interfaz apunta a un objeto de la clase se va llamar al método implementado en la clase.

2. Asociación

Con asociación nos referimos a una relación entre las clases o entre dos objetos. La asociación es una conexión entre clases, que implica la existencia de una relación estructural entre objetos de esas clases. Con asociación tienen sentido las frases “Tiene un”, “Usa un”, “Posee un”. Por ejemplo si tenemos la clase CL_UNIVERSIDAD que tiene un atributo del tipo CL_EMPLEADO, podemos decir que la “Universidad tiene un Empleado”.

```
CLASS cl_empleado DEFINITION.

ENDCLASS.

CLASS cl_universidad DEFINITION.

  PRIVATE SECTION.
    DATA: empleado TYPE REF TO cl_empleado.

ENDCLASS.
```

Los dos objetos pueden existir independientemente, no es una condición de que los objetos se creen a la vez. Son dos objetos que en un momento dado se unen para trabajar juntos. En algún momento el atributo EMPLEADO se va actualizar en la clase CL_UNIVERSIDAD.

3. Composición

Es una relación que representa a objetos compuestos por otros objetos. El objeto en el nivel superior de la jerarquía es el todo y los que están en los niveles inferiores son sus partes o componentes. Un componente es parte esencial de una entidad. La relación es fuerte al punto que si el componente es eliminado o desaparece, la clase mayor (el todo) deja de existir. Por ejemplo, la entidad CL_TELEFONO puede ser modelada en términos de sus componentes de la siguiente forma: “El teléfono tiene una pantalla” o que el teléfono no puede existir si no tiene una pantalla.

Vemos el ejemplo en código:

```
CLASS cl_pantalla DEFINITION.

ENDCLASS.

CLASS cl_telefono DEFINITION.

  PUBLIC SECTION.
    METHODS constructor IMPORTING pantalla TYPE REF TO cl_pantalla.

  PROTECTED SECTION.
    DATA pantalla TYPE REF TO cl_pantalla.

ENDCLASS.

CLASS cl_telefono IMPLEMENTATION.

  METHOD constructor.
    me->pantalla = pantalla.
  ENDMETHOD.

ENDCLASS.
```

La clase CL_TELEFONO no puede existir sin la clase CL_PANTALLA, es una relación entre dos objetos altamente acoplados. Primero tiene que existir una instancia de la clase CL_PANTALLA para instanciar un objeto de la clase CL_TELEFONO.

4. Múltiples referencias apuntando al mismo objeto

Otro concepto que tenemos que tener claro es el de múltiples referencias que apuntan al mismo objeto. Cuando asignamos una referencia de un objeto a otras referencias no se crean en memoria copias de la misma instancia, sino que todas las referencias van a apuntar a la misma instancia y cualquier modificación que efectúa una referencia repercute en todas las demás.

Veamos en un ejemplo creando la clase CL_TARIFAS que tiene el atributo TARIFA_SERVICIO_MEDICO.

```
CLASS cl_tarifas DEFINITION.

  PUBLIC SECTION.
    DATA tarifa_servicio_medico TYPE i.

ENDCLASS.
```

Ahora declaramos tres referencias de la clase creada e instanciamos un objeto en la primera referencia.

```
  DATA: go_1 TYPE REF TO cl_tarifas,
        go_2 TYPE REF TO cl_tarifas,
        go_3 TYPE REF TO cl_tarifas.

  CREATE OBJECT go_1.
```

Asignamos las últimas dos referencias a la referencia que apunta a la instancia creada.

```
  go_2 = go_1.

  go_3 = go_1.
```

Con esta asignación no tenemos en memoria tres instancias, sino que una instancia y tres referencias que apuntan a la misma instancia.

Asignamos un valor al atributo TARIFA_SERVICIO_MEDICO sobre la primera referencia GO_1.

```
  go_1->tarifa_servicio_medico = 30.
```

La modificación sobre la primera referencia afecta a todas las referencias, debido a que solo tenemos una instancia. Con esto queremos decir que el valor del atributo en todas las referencias obtenemos el mismo valor asignado por la primera referencia.

Conclusión: En orientación a objetos si asignamos una referencia a otra referencia no creamos una copia del objeto, sino que las dos referencias van a apuntar a la misma instancia y cualquier modificación que se realiza sobre el objeto afecta a todas las referencias.

5. Crear instancias de tipos distintos

En ABAP no se pueden crear objetos sobre las referencias de las interfaces o de las clases abstractas, pero sobre estas referencias se pueden crear objetos de las clases que han implementado la interfaz o de las subclases que han heredado la clase abstracta.

5.1. Crear instancias de tipos distintos sobre las referencias de las interfaces

Veamos el siguiente ejemplo donde la clase CL_ADMINISTRADOR implementa la interfaz IF_EMPLEADO.

```
INTERFACE if_empleado.

ENDINTERFACE.

CLASS cl_administrador DEFINITION.

  PUBLIC SECTION.
    INTERFACES if_empleado.

ENDCLASS.
```

Como ya sabemos no podemos crear un objeto sobre una referencia de la interfaz.

```
  DATA: go_empleado TYPE REF TO if_empleado.

  go_empleado = NEW ( ).
```

La sentencia NEW resulta en un error de compilación.

Pero en la misma sentencia NEW sobre la misma referencia podemos crear un objeto de una clase que ha implementado la interfaz indicando tipo y el nombre de la clase.

```
go_empleado = NEW cl_administrador( ).
```

La misma sentencia se puede traducir así: una creación del objeto sobre la referencia de la clase y otra de asignación a la referencia de la interfaz.

```
DATA: go_empleado      TYPE REF TO if_empleado,
      go_administrador TYPE REF TO cl_administrador.

go_administrador = NEW ( ).

go_empleado = go_administrador.
```

5.2. Crear instancias de tipos distintos sobre las referencias de las clases

Podemos crear objetos de distinto tipo a la referencia declarada, la única condición clave es que entre la referencia y el tipo indicado tiene que existir una relación de herencia. Sin esta relación es imposible crear el objeto de tipo distinto.

Veamos el siguiente ejemplo donde la clase CL_TECNICO hereda de la clase CL_EMPLEADO.

```
CLASS cl_empleado DEFINITION.

ENDCLASS.

CLASS cl_tecnico DEFINITION INHERITING FROM cl_empleado.

ENDCLASS.
```

Declaramos una referencia de la clase superior.

```
DATA go_empleado TYPE REF TO cl_empleado.
```

Utilizando el nombre de la clase sobre la sentencia NEW para crear en la referencia de la clase superior un objeto de la subclase CL_TECNICO.

```
go_empleado = NEW cl_tecnico( ).
```

6. Asignar instancias utilizando la clase genérica OBJECT

En ABAP existe la clase genérica OBJECT, que es un tipo predefinido. Aunque no tenemos una relación directa, cualquier clase que se crea es también del tipo OBJECT. Normalmente decimos que una subclase “es un tipo” de la superclase, por ejemplo un león es un animal donde la clase LEON hereda de la clase ANIMAL. Lo mismo se puede decir sobre todas las clases, aún que no hereda directamente de la clase OBJECT es un tipo OBJECT. Teniendo en cuenta esta consideración podemos crear referencias genéricas y asignar cualquier tipo de cualquier clase a las referencias de tipo OBJECT.

Veamos el siguiente ejemplo donde en una referencia de la clase OBJECT asignamos una instancia de la clase CL_WEB_SERVICE.

```
CLASS cl_web_service DEFINITION.

  PUBLIC SECTION.
    METHODS m1.

ENDCLASS.

CLASS cl_web_service IMPLEMENTATION.

  METHOD m1.
    WRITE 'Hola'.
  ENDMETHOD.

ENDCLASS.
```

Referencia a la clase generica OBJECT.

```
  DATA go_object TYPE REF TO object.
```

Instanciamos utilizando el añadido TYPE en la referencia del OBJECT un objeto de la clase CL_WEB_SERVICE.

```
  go_object = NEW cl_web_service( ).
```

Para llamar a los métodos del objeto por la referencia de la clase OBJECT tenemos que utilizar llamadas dinámicas porque dentro de la clase OBJECT no existe la implementación del método M1, pero sabemos que el objeto creado tiene el método M1. Entre paréntesis se puede pasar el nombre del método entre comillas simples.

```
go_object->('M1').
```

También se puede pasar una variable que contiene el nombre de método, por ejemplo:

```
DATA gv_method_name TYPE string.

gv_method_name = 'M1'.

CALL METHOD gr_object->(gv_method_name).
```

Si el método tiene parámetros IMPORTING, EXPORTING, RETURNING, etc. se tienen que pasar en cuando se realiza la llamada dinámica.

```
CALL METHOD gr_object->(gv_method_name)
     EXPORTING nombre_parametro = valor_parametro.
```

Otra forma para realizar llamadas dinámicas a los métodos de una clase es utilizando el tipo dinámico ABAP_PARMBIND_TAB que es una tabla interna hashed con tres campos:

- NAME – se tiene que informar con el nombre del parámetro
- KIND – Es un CHAR con una longitud de uno y se tiene que informar con el tipo de parámetro. Los valores que admite son E para los parámetros de tipo EXPORTING, C para CHANGING y R para RETURNING y RECIEVING.
- VALUE – se tiene que informar con el valor que se le va pasar al parámetro.

La tabla interna debe que contener un registro por cada parámetro que tiene el método. Para realizar la llamada dinámica con la tabla interna se utiliza el añadido PARAMETER-TABLE del siguiente modo:

```
CALL METHOD gr_object->(gv_method_name) PARAMETER-TABLE gt_param.
```

Donde GT_PARAM es la tabla interna del tipo ABAP_PARMBIND_TAB.