LOGALI

TEORÍA

Herencia, Casting y Clase Amiga

SAP ABAP Programación Orientada a Objetos

Contenido

1. Herencia

1.1. Conceptos

En programación orientada a objetos, la herencia es el mecanismo más utilizado para alcanzar algunos de los objetivos más preciados en el desarrollo de software como lo son la reutilización y la extensibilidad. A través de herencia los diseñadores pueden crear nuevas clases partiendo de una clase o de una jerarquía de clases preexistente (ya comprobadas y verificadas) evitando con ello el rediseño, la modificación y verificación de la parte ya implementada. La herencia facilita la creación de objetos a partir de otros ya existentes e implica que una subclase obtenga todo el comportamiento (métodos) y eventualmente los atributos (variables) de su superclase.

La herencia también es la relación entre una clase general y otra clase más específica. Por ejemplo, si declaramos una clase COCHE derivada de una clase VEHICULO, todos los métodos y variables asociadas con la clase vehículo, son automáticamente heredados por la subclase coche.

La herencia es uno de los mecanismos de los lenguajes de programación orientada a objetos basados en clases, por medio del cual una clase se deriva de otra de manera que extiende su funcionalidad. La clase de la que se hereda se suele denominar *clase base*, *clase padre* o *superclase*.

1.2. Ventajas

- Ayuda a los programadores a ahorrar código y tiempo, ya que la clase padre ha sido implementada y verificada con anterioridad, restando solo referenciar desde la clase derivada a la clase base.
- Los objetos pueden ser construidos a partir de otros similares. Para ello es necesario que exista una clase base (que incluso puede formar parte de una jerarquía de clases más amplia).
- La clase derivada hereda el comportamiento y los atributos de la clase base, y es común que se le añada su propio comportamiento o que modifique lo heredado.
- Toda clase pueden servir como clase base para crear otras.

1.3. Desventajas

Si la jerarquía de clases es demasiado compleja, el programador puede tener problemas para comprender el funcionamiento de un programa. Además puede volverse más complejo detectar y resolver errores de programación, por ejemplo al modificar una clase padre que afecta el funcionamiento de las subclases.

1.4. Estereotipos de herencia

1.4.1. Herencia simple

Una clase sólo puede heredar de una clase base y de ninguna otra clase más.

1.4.2. Herencia múltiple

Una clase puede heredar las características de varias clases base, es decir, puede tener varios padres. En este aspecto hay discrepancias entre los diseñadores de lenguajes. Algunos de ellos han preferido no admitir la herencia múltiple debido a que los potenciales conflictos entre métodos y variables con igual nombre, y eventualmente con comportamientos diferentes crea un desajuste cognitivo que va en contra de los principios de la programación orientada a objetos. Por ello, la mayoría de los lenguajes orientados a objetos admite herencia simple. **ABAP solo admite herencia simple** (y resuelve la herencia múltiple con las interfaces).

1.5. Herencia – Implementación

La herencia permite crear una nueva clase a partir de una nueva existente heredando la nueva clase sus propiedades. Esto se realiza añadiendo la adición **INHERITING FROM** a la sentencia de definición de la clase:

```
CLASS <subclass> DEFINITION INHERITING FROM <superclass>.
```

La nueva clase **<subclass>** hereda todos los componentes de la clase ya existente **<superclase>**.

La nueva clase se conoce como la subclase de la clase de la que procede. La clase original se conoce como la superclase de la nueva clase. Si no se añade ninguna declaración a la subclase, esta contiene los mismos componentes que la superclase. De cualquier manera, sólo los componentes públicos y protegidos de la superclase son visibles en la subclase. Aunque los componentes privados de la superclase existen en la subclase, no son visibles. Se pueden declarar componentes privados en una subclase que tengan los mismos nombres que componentes privados de la superclase. Cada clase trabaja con sus propios componentes privados.

1.6. Redefinición de métodos

Todas las subclases contienen los componentes de todas las clases existentes entre ellas mismas y el nodo raíz del árbol de herencia. La visibilidad de un componente no puede ser cambiada nunca. En cambio se puede usar la adición **REDEFINITION** en la sentencia **METHODS** para redefinir un método público o protegido dependiente de instancia en una subclase y hacer que realice una función más especializada. Cuando se redefine un método no se puede cambiar su interface, el método mantiene el mismo nombre y la misma interface de parámetros, pero tiene una nueva implementación. La declaración y la implementación de un método en una superclase no se ven afectados cuando se redefine un método en una subclase. La implementación de la redefinición en la subclase 'oculta' la implementación original en la superclase. Cualquier referencia que apunte a un objeto de la subclase usa el método redefinido, incluso si la referencia fue definida con referencia a la superclase. Esto se aplica particularmente a la referencia a sí mismo me->.

Si por ejemplo un método M1 de una superclase contiene una llamada CALL METHOD [ME->]M2 y M2 está redefinido en una subclase, la llamada a M1 desde una instancia de la superclase hará que el método original M2 sea llamado, mientras que la llamada a M1 desde una instancia de la subclase hará que el método redefinido M2 sea llamado (llaman a 'distintos' métodos, aunque tengan el mismo nombre). Dentro de un método redefinido se puede usar la referencia SUPER-> para acceder al método 'oculto'. Esto permite usar la funcionalidad existente en el método de la superclase sin tener que codificarla de nuevo en la subclase.

2. Casting

2.1. Narrowing Cast

Cuando asignamos la instancia de la superclase a la subclase se llama Narrowing Cast porque nos estamos moviendo de la "Vista más específica" a la "Vista menos específica".

No es posible mover siempre la referencia de la superclase a la subclase, porque la subclase podría tener más funcionalidad en comparación con la superclase. Por lo tanto, tenemos que ser más cuidadosos cuando usamos Narrowing Cast. El error se puede capturar con CX_SY_MOVE_CAST_ERROR.

2.2. Widening Cast

Las variables de referencia de la clase superior también pueden hacer referencia a las instancias de subclase en tiempo de ejecución. Puede copiar esta referencia nuevamente a una variable de referencia del tipo de subclase. Para asignar una referencia de clase superior a una referencia de subclase, debe utilizar el operador de asignación Widening Cast:

MOVE ... ?TO ... o su forma breve ?=.

Sino, obtendrá un mensaje indicando que no es seguro que todos los componentes a los que se puede acceder sintácticamente después de la asignación cast estén disponibles en la actualidad en la instancia. Como regla, la clase de subclase contiene más componentes que la clase superior.

3. Concepto FINAL

3.1. Clase final

Una clase final es la última del árbol de herencia y no puede tener subclases. Si se crea una clase abstracta final sólo se puede acceder a sus componentes estáticos. Se aplica añadiendo la palabra clave FINAL a una clase.

```
CLASS NOMBRE_CLASE DEFINITION FINAL.
```

3.2. Método final

Si la creación de clases finales no cumple con nuestras necesidades, y realmente lo que se quiere es proteger algunos métodos de una clase para que no sean redefinidos, se puede utilizar las palabras claves FINAL REDEFINITION en la declaración de método para indicar al compilador que este método no puede ser redefinido por las subclases.

```
METHODS NOMBRE_METODO FINAL REDEFINITION.
```

4. Encapsular instancias

La creación de instancias tiene los mismos tipos de encapsulación que tenemos para los componentes de la clase: pública, protegida y privada. La sentencia NEW para instanciar un objeto se puede utilizar en el ámbito de visibilidad que permite la encapsulación de instancia declarada en la definición de la clase.
Si no indicamos la encapsulación de la instancia en cuando definimos una clase el sistema le asigna el encapsulamiento público por defecto.

4.1. Encapsulación pública para la creación de las instancias

```
CLASS nombre_clase DEFINITION CREATE PUBLIC.
```

Se pueden crear instancias utilizando NEW fuera de la clase, en las subclases y dentro de la clase.

4.2. Encapsulación protegida para la creación de las instancias

```
CLASS nombre_clase DEFINITION CREATE PROTECTED.
```

Se pueden crear instancias utilizando NEW en las subclases y dentro de la clase.

4.3. Encapsulación privada para la creación de las instancias

```
CLASS nombre_clase DEFINITION CREATE PRIVADE.
```

Se pueden crear instancias utilizando NEW solo dentro de la clase.

5. Concepto Friends

En cualquier lenguaje de programación orientado a objetos, el acceso a los componentes privados o protegidos está prohibido tanto para los métodos como para los atributos. Si alguien trata de acceder a ellos, el compilador genera un error de sintaxis. A veces, sería ventajoso para dar el acceso a estos atributos protegidos y privados a otras clases. Esto se puede lograr usando la adición de FRIENDS.

5.1. Diseño y Consideraciones

Esta adición tiene que ser añadida en la clase cuyos componentes deben ser visualizados. Creamos la clase MI_CLASE que es la clase que va proporcionar el acceso a sus atributos privados a otra clase. La clase que quiere acceder a los atributos de la clase MI_CLASE no necesita ninguna declaración especial, por lo tanto creamos la clase AMIGO como clase que accede a los atributos privados de la clase MI_CLASE.

```
CLASS MI_CLASE DEFINITION FRIENDS AMIGO.
```

5.2. Las subclases de FRIENDS

Todas las subclases de la clase AMIGO (AMIGO_HIJOS) serían por defecto FRIENDS de la clase que ofrece la amistad (MI_CLASE) y tendrían acceso ilimitado a todos los componentes.