

Programación Orientada a Objetos OOP IEC61131-3 Curso Youtube by Runtimevic

**Programación Orientada a Objetos OOP IEC61131-3 PLC Curso Youtube
by Runtimevic.**

runtimevic

Copyright © 2023 Víctor Durán.

Table of contents

1. Requisitos	4
2. Introduccion	5
3. Tipos de paradigmas	7
4. Clases y Objetos	9
4.1 Clases y Objetos	9
4.2 Bloque de Funciones	11
4.3 Objeto Metodo	16
4.4 Objeto Propiedad	21
4.5 Herencia	23
4.6 THIS puntero	28
4.7 SUPER puntero	31
4.8 Interface	33
4.9 puntero y referencia	35
4.10 FB abstracto frente a interfaz	42
5. Tabla de Modificadores de acceso	43
6. Tipos de variables y variables especiales	44
7. Tipos de Datos	48
7.1 Declaracion de una Variable:	48
8. Interfaz fluida	50
9. Principios OOP	51
9.1 4 Pilares	51
9.2 Abstracción	52
9.3 Encapsulamiento	53
9.4 Herencia	54
9.5 Polimorfismo	55
10. Relaciones	56
11. SOLID	57
11.1 SOLID	57
11.2 Principio de Responsabilidad Única	58
11.3 Principio de Abierto/Cerrado	59
11.4 Principio de Sustitución de Liskov	60
11.5 Principio de Segregación de Interfaz	61
11.6 Principio de Inversión de Dependencia	62
12. UML	63
12.1 UML	63

12.2 Class UML	64
12.3 StateChart UML	65
13. Tipos de Diseño para programacion de PLC	66
14. Patrones de Diseño	67
14.1 Patrones de Diseño	67
14.2 Patron de Estrategia	73
15. Links	74

1. Requisitos

👤 Requisitos 🤖:



Los requisitos necesarios para seguir este curso serían tener instalados los siguientes softwares:

- [Beckhoff TwinCAT 3 XAE](#) ó el IDE de [Codesys](#).
- Tener cuenta de usuario creada en [GitHub](#).
- saber lo mínimo de Git o apoyarse en herramientas visuales como pueden ser:
- [GitHub Desktop](#).
- [sourcetree](#)
- [tortoiseGit](#), etc...
- Sería bueno tener algo de conocimientos previos de teoria de OOP, aunque sean en otros lenguajes de programación ya que seran extrapolables para el enfoque de este curso de OOP IEC61131-3 para PLCs.

1.0.1 Pasos para empezar:

- Clonar el repositorio de [GitHub](#):

```
$ git clone https://github.com/runtimevic/OOP-IEC61131-3--Curso-Youtube.git
```

 ó utilizar por ejemplo GitHub Desktop para Clonar el repositorio de GitHub...
- Nos encontraremos las siguientes carpetas:
- [TC3_OOP](#): Dentro de esta carpeta se encuentra el proyecto de TwinCAT3, con todo lo que se va explicando en los videos de youtube...
- [Ficheros_PLCOpen_XML](#): Dentro de esta carpeta nos iremos encontrando los ficheros exportados en formato PLCOpen XML para que puedan ser importados en TwinCAT3 ó en Codesys de todo lo explicado en Youtube, ya que al ser el formato standarizado de PLCOpen se puede exportar/importar en todas las marcas de PLCs que sigan el estandard PLCOpen..., pero es recomendable intentar realizar lo explicado desde cero para ir practicando y asumir los conceptos explicados...
- tambien esta alojada la creación de esta pagina web SSG, (Generador de Sitios Estáticos) la cual se ira modificando conforme avancemos en este Curso de OOP IEC-61131-3 PLC...

2. Introduccion

📖 Curso Programación Orientada a Objetos Youtube -- OOP :



by Runtimevic -- Víctor Durán Muñoz.

2.0.1 ¿ Qué es OOP?

- Es un paradigma que hace uso de los objetos para la construcción de los software.
- ¿ Qué es un paradigma?
- Tiene diferentes interpretaciones, puede ser un **modelo**, **ejemplo** o **patrón**.
- Es una **forma** o un **estilo** de programar.
- se busca plasmar la realidad hacia el código.

2.0.2 ¿Cómo pensar en Objetos?

- Enfocarse en **algo de la realidad**.
- Detalla sus **atributos**, (**propiedades**)

- Detalla sus **comportamientos (metodos)**

```
1  📱 Ejemplo: (Telefono móvil-smartphone)
2
3  . ¿Qué atributos (Propiedades) reconocemos?
4      - color.
5      - marca.
6  . ¿Qué se puede hacer? (Metodos)
7      - Realizar llamadas.
8      - Navegar por internet.
```

```
1  🚗 Ejemplo: (Coche)
2
3  . ¿Qué atributos (Propiedades) reconocemos?
4      - color.
5      - marca.
6  . ¿Qué se puede hacer? (Metodos)
7      - conducir.
8      - frenar.
9      - acelerar.
```

2.0.3 Links:

- [🔗 Codesys admite OOP](#)
- [🔗 Beckhoff TwinCAT 3 admite OOP](#)

3. Tipos de paradigmas

3.0.1 Tipos de paradigmas:

- Imperativa -- (**Instrucciones a seguir** para dar solución a un problema).
- Declarativa -- (Se **enfoca en el problema** a solucionar).
- Estructurada -- (La solución a un problema sigue **una secuencia de inicio a fin**).
- Funcional -- (Divide el problema en diversas soluciones que serán ejecutadas por las **funciones declaradas**). La programación procedimental o programación por procedimientos es un paradigma de la programación. Muchas veces es aplicable tanto en lenguajes de programación de bajo nivel como en lenguajes de alto nivel. En el caso de que esta técnica se aplique en lenguajes de alto nivel, recibirá el nombre de programación funcional.
- se llaman rutinas separadas desde el programa principal
- datos en su mayoría globales -> sin protección.
- los procedimientos por lo general no pueden ser independientes -> mala reutilización del código.



- **Orientada a objetos** -- Construye soluciones **basadas en objetos**.

```

1  wikipedia:
2  La programación orientada a objetos es un paradigma de programación
3  basado en el concepto de "objetos", que pueden contener datos y código.
4  Los datos están en forma de campos y el código está en forma de procedimientos.

```



3.0.2 Ventajas de la Programación OOP:

- rutinas y datos se combinan en un objeto -> Encapsulación.
- métodos/Propiedades -> interfaces definidas para llamadas y acceso a datos.

4. Clases y Objetos

4.1 Clases y Objetos

Clases y Objetos:

- Una Clase es una **plantilla**.
- Un Objeto es la **instancia de una Clase**.



```
1 En este Ejemplo Nos encontramos la Clase Coche,  
2 y hemos instanciado esta Clase para tener los Objetos de Coches  
3 Mercedes, Bmw y Audi...
```

Representacion de la Clase Coche en STL OOP IEC 61131-3

```

1  FUNCTION_BLOCK Coche
2  VAR_INPUT
3  END_VAR
4  VAR_OUTPUT
5  END_VAR
6  VAR
7      _Marca : STRING;
8      _Color : STRING;
9      accion : STRING;
10 END_VAR
11 -----
12 METHOD PUBLIC Acelerar
13 accion := 'acelerar';
14 -----
15 METHOD PUBLIC Conducir
16 accion := 'conducir';
17 -----
18 METHOD PUBLIC Frenar
19 accion := 'frenar';
20 -----
21 PROPERTY PUBLIC Color : STRING
22 Get
23     Color := _Color;
24 Set
25     _Color := Color;
26 -----
27 PROPERTY PUBLIC Marca : STRING
28 Get
29     Marca := _Marca;
30 Set
31     _Marca := Marca;

```

Instancia de la clase en los Objetos: Mercedes,Bmw y Audi y llamadas a sus metodos y propiedades...

```

1  PROGRAM _01_Clase_y_Objetos
2  VAR
3      // tenemos la Clase Coche y la instanciamos y obtenemos los Objetos: Mercedes, Bmw y Audi.
4      Mercedes : Coche;
5      Bmw : Coche;
6      Audi: Coche;
7
8      Color : STRING;
9      Marca : STRING;
10
11     Acelerar : BOOL;
12     Conducir: BOOL;
13     Frenar : BOOL;
14 END_VAR
15
16 //Objeto Mercedes
17 //llamadas a sus métodos.
18 IF Acelerar THEN
19     Mercedes.Acelerar();
20     Acelerar := FALSE;
21 END_IF
22
23 IF Conducir THEN
24     Mercedes.Conducir();
25     Conducir := FALSE;
26 END_IF
27
28 IF Frenar THEN
29     Mercedes.Frenar();
30     Frenar := FALSE;
31 END_IF
32
33 //llamadas a sus propiedades.
34 Mercedes.Marca := 'Mercedes';
35 Mercedes.Color := 'Negro';
36 Color := Mercedes.Color;

```

Links:

- [🔗 methods-properties-and-inheritance \(stefanhenneken\)](#)

4.2 Bloque de Funciones

4.2.1 Bloques de Funciones

DECLARACION DE UN FUNCTION BLOCK:

```
1 FUNCTION_BLOCK <access specifier> <function block> | EXTENDS <function block> | IMPLEMENTS <comma-separated list of interfaces>
```

IMPLEMENTACIÓN BLOQUE DE FUNCIONES:

. Bloque de Funciones:

- Representa la Clase.
- Intercambio de datos por variables:
Entrada, Salida, Entrada/Salida
- Encapsulación de datos por:
Variables locales, Propiedades.
- Ejecución por Metodos.
- Construcción y Destrucción de Objetos
por Constructor, Destructor.



EXTENDS: - Si en la declaración de un FUNCTION_BLOCK añadimos la palabra EXTENDS seguida del nombre del FB del cual queremos heredar, significa que heredamos todos sus metodos y propiedades.(principio de Herencia) - Un FB solo puede heredar de una Clase FB.

IMPLEMENTS: - Si en la declaración de un FUNCTION_BLOCK añadimos la palabra IMPLEMENTS seguido del nombre de la interfaz o interfaces separadas por comas. - Si en el FB se implementa una interfaz es obligatorio en el FB crear la programación de los metodos y propiedades de la interfaz implementada.

- Ejemplos de declaración de FUNCTION_BLOCK:

```
1 FUNCTION_BLOCK INTERNAL ABSTRACT FB
2 FUNCTION_BLOCK INTERNAL FINAL FB
3 FUNCTION_BLOCK PUBLIC FINAL FB
4 FUNCTION_BLOCK ABSTRACT FB
5 FUNCTION_BLOCK PUBLIC ABSTRACT FB
6 FUNCTION_BLOCK FB EXTENDS FB1 IMPLEMENTS Interface1, Interface2, Interface3
```

4.2.2 Bloque de Funcion Modificadores de acceso

MODIFICADORES DE ACCESO BLOQUE DE FUNCIONES:

Podemos tener 2 modificadores de acceso para el Bloque de Funciones:

- **PUBLIC:**

- No hay restricciones, se puede llamar desde cualquier lugar.
- Si no ponemos nada al declarar el FB es lo mismo que PUBLIC.
- Cualquiera puede llamar o crear una instancia del FB.
- Se puede usar para la herencia al ser public.
- Son accesibles luego de instanciar la clase.
- Corresponde a la especificación de modificador sin restricción de acceso.

- **INTERNAL:**

- Solo se puede acceder al FB desde el mismo espacio de nombres.
- Esto permite que el FB este disponible solo dentro de una determinada biblioteca. La configuración predeterminada donde no se define ningún especificador de acceso es PUBLIC .
- El acceso está limitado al espacio de nombres (la biblioteca).

Podemos tener otros 2 modificadores de acceso para el Bloque de Funciones: - **FINAL:** - (en TwinCAT 3 no sale por defecto para seleccionarlo al crear un FB, pero se puede añadir mas tarde despues de crearlo...) - El FB no puede servir como un bloque de funciones principal. - Los métodos y las propiedades de esta POU no se pueden heredar. - FINAL solo está permitido para POU del tipo FUNCTION_BLOCK. - No se permite sobrescribir, en un derivado del bloque de funciones. - Esto significa que no se puede sobrescribir/extender en una subclase posiblemente existente.

- **ABSTRACT:** bloques de funciones abstractas

```
1 FUNCTION_BLOCK PUBLIC ABSTRACT FB_Foo
```

- Tan pronto como un método o una propiedad se declaran como abstractos , el bloque de funciones también debe declararse como abstracto . - No se pueden crear instancias a partir de FB abstractos. Los FB abstractos solo se pueden usar como FB básicos cuando se heredan. - Todos los métodos abstractos y todas las propiedades abstractas deben sobrescribirse para crear un FB no abstracto. Un método abstracto o una propiedad abstracta se convierte en un método no abstracto o una propiedad no abstracta al sobrescribir. - Los bloques de funciones abstractas pueden contener además métodos no abstractos y/o propiedades no abstractas. - Si no se sobrescriben todos los métodos abstractos o todas las propiedades abstractas durante la herencia, el FB heredado solo puede ser un FB abstracto (concretización paso a paso). - Se permiten punteros o referencias de tipo FB abstracto. Sin embargo, estos pueden referirse a FB no abstractos y, por lo tanto, llamar a sus métodos o propiedades (polimorfismo).

4.2.3 Bloque de Funcion Declaracion de variables

TIPOS DE VARIABLES QUE SE PUEDEN DECLARAR EN UN FUNCTION_BLOCK:

- [Local Variables - VAR](#)
- [Input Variables - VAR_INPUT](#)
- [Output Variables - VAR_OUTPUT](#)
- [Input/Output Variables - VAR_IN_OUT, VAR_IN_OUT CONSTANT](#)
- [Temporary Variable - VAR_TEMP](#)
- [Static Variables - VAR_STAT](#)
- [External Variables - VAR_EXTERNAL](#)
- [Instance Variables - VAR_INST](#)
- [Remanent Variables - PERSISTENT, RETAIN](#)
- [SUPER](#)
- [THIS](#)
- [Variable types - attribute keywords](#)
- [RETAIN: for remanent variables of type RETAIN](#)
- [PERSISTENT: for remanent variables of type PERSISTENT](#)
- [CONSTANT: for constants](#)
- Todos estos tipos de variables que se pueden declarar dentro del FB se pueden repetir los mismos tipos de variables dentro del FB, esto podria valer para diferenciar variables del mismo tipo en la zona de declaración, sería meramente indicativo...
- Ejemplo de declaración de variables en un **FUNCTION_BLOCK**:

```

1  FUNCTION_BLOCK fb_tipos_de_datos
2  VAR_INPUT
3      binput : BOOL;
4  END_VAR
5  VAR_INPUT
6      binput2 : BOOL;
7  END_VAR
8  VAR_OUTPUT
9      output1 : REAL;
10 END_VAR
11 VAR_IN_OUT
12     in_out1 : LINT;
13 END_VAR
14 VAR_IN_OUT CONSTANT
15     in_out_constant1 : DINT;
16 END_VAR
17 VAR
18     var1 : STRING;
19 END_VAR
20 VAR_TEMP
21     temp1 : ULINT;
22 END_VAR
23 VAR_STAT
24     nVarStat1 : INT;
25 END_VAR
26 VAR_EXTERNAL
27     nVarExt1 : INT; // 1st external variable
28 END_VAR
29 VAR PERSISTENT
30     nVarPers1 : DINT; (* 1. Persistent variable *)
31     bVarPers2 : BOOL; (* 2. Persistent variable *)
32 END_VAR
33 VAR RETAIN
34     nRem1 : INT;
35 END_VAR
36 VAR CONSTANT
37     n : INT := 10;
38 END_VAR

```

4.2.4 Constructor y Destructor

MÉTODOS 'FB_INIT', 'FB_REINIT' Y 'FB_EXIT':

FB_INIT:

- Dependiendo de la tarea, puede ser necesario que los bloques de funciones requieran parámetros que solo se usan una vez para las tareas de inicialización. Una forma posible de pasarlos elegantemente es usar el método FB_init(). Este método se ejecuta implícitamente una vez antes de que se inicie la tarea del PLC y se puede utilizar para realizar tareas de inicialización.
- También es posible sobrescribir FB_init(). En este caso, las mismas variables de entrada deben existir en el mismo orden y ser del mismo tipo de datos que en el FB básico. Sin embargo, se pueden agregar más variables de entrada para que el bloque de funciones derivado reciba parámetros adicionales.
- Al pasar los parámetros por FB_init(), no se pueden leer desde el exterior ni cambiar en tiempo de ejecución. La única excepción sería la llamada explícita de FB_init() desde la tarea del PLC. Sin embargo, esto debe evitarse principalmente, ya que todas las variables locales de la instancia se reinicializarán en este caso. Sin embargo, si aún debe ser posible el acceso, se pueden crear las propiedades apropiadas para los parámetros.

FB_REINIT:

Si es necesario, debe implementar FB_reinit explícitamente. Si este método está presente, se llama automáticamente después de que se haya copiado la instancia del bloque de función correspondiente (llamada implícita). Esto sucede durante un cambio en línea después de cambios en la declaración de bloque de función (cambio de firma) para reinicializar el nuevo bloque de instancia. Este método se llama después de la operación de copia y debe establecer valores definidos para las variables de la instancia. Por ejemplo, puede inicializar variables en consecuencia en la nueva ubicación en la memoria o notificar a otras partes de la aplicación sobre la nueva ubicación de variables específicas en la memoria. Diseñe la implementación independientemente del cambio en línea. El método también se puede llamar desde la aplicación en cualquier momento para restablecer una instancia de bloque de funciones a su estado original.

FB_EXIT:

Si es necesario, debe implementar FB_exit explícitamente. Si este método está presente, se llama automáticamente (implícitamente) antes de que el controlador elimine el código de la instancia del bloque de funciones (por ejemplo, incluso si TwinCAT cambia del modo Ejecutar al modo de configuración).

LINKS:

Caso operativo "Primera descarga"	Caso operativo "Nueva descarga"	Caso operativo "Online Change"
1. FB_init (código de inicialización implícito y explícito) 2. Inicialización explícita de variables externas mediante declaración de instancia del bloque de funciones 3. Método declarado con el atributo 'call_after_init'	1. FB_exit 2. FB_init (código de inicialización implícito y explícito) 3. Inicialización explícita de variables externas mediante declaración de instancia del bloque de funciones 4. Método declarado con el atributo 'call_after_init'	1. FB_exit 2. FB_init (código de inicialización implícito y explícito) 3. Inicialización explícita de variables externas mediante declaración de instancia del bloque de funciones 4. Método declarado con el atributo 'call_after_init' 5. Procedimiento de copia 6. FB_reinit
Parámetros del método: FB_init(bInitRetains := TRUE, bInCopyCode := FALSE);	Parámetros del método: FB_exit(bInCopyCode := FALSE); FB_init(bInitRetains := TRUE, bInCopyCode := FALSE);	Parámetros del método: FB_exit(bInCopyCode := TRUE); FB_init(bInitRetains := FALSE, bInCopyCode := TRUE);

- [Métodos FB_init, FB_reinit and FB_exit, Infosys Beckhoff](#)
- [Métodos 'FB_Init', 'FB_Reinit' y 'FB_Exit', Codesys](#)
- [iec-61131-3-parameter-transfer-via-fb_init, stefanhenneken.net](#)

4.3 Objeto Metodo

4.3.1 Metodo

METHOD:

Los Métodos dividen la clase (bloque de funciones) en funciones más pequeñas que se pueden ejecutar en llamada. Solo trabajarán con los datos que necesitan e ignorarán cualquier dato redundante que puede existir en un determinado bloque de funciones.

Los métodos pueden acceder y manipular las variables internas de la clase principal, pero también pueden usar variables propias a las que la clase principal no puede acceder (a menos que sean de salida la variable).

Además, los métodos son una forma mucho más eficiente de ejecutar un programa porque, al dividir una función en varios métodos, el usuario evita ejecutar todo el POU cada vez, ejecutar solo pequeñas porciones de código siempre que sea necesario llamarlas.

Esto es un muy buena manera de evitar errores y corrupción de datos. Los métodos también tienen un nombre, lo que significa que estas porciones de código se pueden identificar por su propósito en lugar de las variables que manipulan, mejorando así la lectura de código, comprensión y la solución de problemas.

La abstracción juega un papel importante aquí, si los programadores desean implementar el código, solo necesitan llamar al método.

La solución de problemas también se convierte en más simple: entonces el programador no necesita buscar cada instancia del código, solo necesitan verificar el método correspondiente. A diferencia de la clase base, los métodos usan la memoria temporal del controlador: los datos son volátiles, ya que las variables solo mantendrán sus valores mientras se ejecuta el método. Si se suponen valores que deben mantenerse entre ciclos de ejecución, entonces la variable debe almacenarse en la clase base o en algún otro lugar que retendrá los valores de un ciclo al otro (como la lista de variables globales -- GVL), o también se puede utilizar la variable de tipo VAR_INST.

Por lo tanto, una declaración de Método tiene la siguiente estructura:

```
1 METHOD <Access specifier> <Name> : <Datatype return value>
```

No es obligatorio que un Método deba devolver un valor...

EJEMPLO DE DECLARACIÓN DE METHOD:

```
1 METHOD Method1 : BOOL
2 VAR_INPUT
3   nIn1 : INT;
4   bIn2 : BOOL;
5 END_VAR
6 VAR_OUTPUT
7   fOut1 : REAL;
8   sOut2 : STRING;
9 END_VAR
```

LINKS DEL OBJETO METODO:

- [Documentación Codesys del Objeto método](#)
- [Documentación de Beckhoff del Objeto método](#)
- [TC08.Beckhoff TwinCAT3 Function Block-Part3 Method.JP](#)

4.3.2 Metodo Modificadores de acceso

ESPECIFICADORES DE ACCESO PARA LOS METODOS:

La declaración del método puede incluir un especificador de acceso opcional. Esto restringe el acceso al método.

TIPOS DE MODIFICADORES DE ACCESO PARA EL MÉTODO:

- **PUBLIC:**
 - Cualquiera puede llamar al método, no hay restricciones.
- **PRIVATE:**
 - El método está disponible solo dentro de la POU. No se puede llamar desde fuera de la POU.
 - Son accesibles dentro de la clase.
 - El acceso está restringido al bloque de funciones o al programa, respectivamente.
- **PROTECTED:**
 - Solo su propia POU o las POU derivadas (herencia) de ella pueden acceder al método. La derivación se analiza a continuación.
 - Son accesibles a través de la herencia.
 - El acceso está restringido al programa o al bloque de función y sus derivados respectivamente.
- **INTERNAL:**
 - Solo se puede acceder al método desde el mismo espacio de nombres. Esto permite que los métodos estén disponibles solo dentro de una determinada biblioteca, por ejemplo.
 - El acceso está limitado al espacio de nombres (la biblioteca).

La configuración predeterminada donde no se define ningún especificador de acceso es PUBLIC .

- **FINAL:(se puede añadir acompañado con alguno de los anteriores)**
- El método no puede ser sobrescrito por otro método. La sobrescritura de métodos se describe a continuación.
- No se permite sobrescribir, en un derivado del bloque de funciones.
- Esto significa que no se puede sobrescribir/extender en una subclase posiblemente existente.

4.3.3 Metodo Declaracion de variables

TIPOS DE VARIABLES QUE SE PUEDEN DECLARAR EN UN METHOD:

- [Local Variables - VAR](#)
- [Input Variables - VAR_INPUT](#)
- [Output Variables - VAR_OUTPUT](#)
- [Input/Output Variables - VAR_IN_OUT, VAR_IN_OUT CONSTANT](#)
- [Temporary Variable - VAR_TEMP](#)
- [Static Variables - VAR_STAT](#)
- [External Variables - VAR_EXTERNAL](#)
- [Instance Variables - VAR_INST](#)
- [Remanent Variables - PERSISTENT, RETAIN](#)
- [SUPER](#)
- [THIS](#)
- [Variable types - attribute keywords](#)
- [RETAIN: for remanent variables of type RETAIN](#)
- [PERSISTENT: for remanent variables of type PERSISTENT](#)
- [CONSTANT: for constants](#)
- Ejemplo de declaración de variables en un **METHOD**:

```

1  METHOD metodo0_Declaracion_variables
2  VAR_INPUT
3      binput : BOOL;
4  END_VAR
5  VAR_INPUT
6      binput2 : BOOL;
7  END_VAR
8  VAR_OUTPUT
9      output1 : REAL;
10 END_VAR
11 VAR_IN_OUT
12     in_out1 : LINT;
13 END_VAR
14 VAR_IN_OUT CONSTANT
15     in_out_constant1 : DINT;
16 END_VAR
17 VAR
18     var1 : STRING;
19 END_VAR
20 //!!! no se pueden declarar variables TEMPORALES dentro de la zona de declaración de variables del método!!!
21 //VAR_TEMP
22 //     temp1 : ULINT;
23 //END_VAR
24 VAR_INST
25     counter : INT;
26 END_VAR
27 VAR_STAT
28     nVarStat1 : INT;
29     aarray : ARRAY[1..n] OF INT;
30 END_VAR
31 VAR_EXTERNAL
32     nVarExt1 : INT; // 1st external variable
33 END_VAR
34 //!!! no se pueden declarar variables PERSISTENT ni RETAIN dentro de la zona de declaración de variables del método!!!
35 //VAR PERSISTENT
36 //     nVarPers1 : DINT; (* 1. Persistent variable *)
37 //     bVarPers2 : BOOL; (* 2. Persistent variable *)
38 //END_VAR
39 //VAR RETAIN
40 //     nRem1 : INT;
41 //END_VAR
42 VAR CONSTANT
43     n : INT:= 10;
44 END_VAR

```

4.3.4 Metodo tipos de variables de retorno

TIPOS DE VARIABLES DE RETORNO:

- No es obligatorio en el metodo retornar un tipo de variable.
- Ejemplos de declaración de Métodos que nos devuelve una variable de diferentes tipos:

```
1 METHOD Method1 : BOOL
2 METHOD Method1 : INT
3 METHOD Method1 : REAL
4 METHOD Method1 : STRING
```

RETORNO POR STRUCT:

Acceso a un único elemento de un tipo de retorno estructurado durante la llamada a método/función/propiedad.

La siguiente implementación se puede utilizar para tener acceso directamente a un elemento individual del tipo de datos estructurado que devuelve el método/función/propiedad cuando se llama a un método, función o propiedad.

Un tipo de datos estructurado es, por ejemplo, una estructura o un bloque de funciones.

El tipo devuelto del método/función/propiedad se define como:

```
1 REFERENCE TO <structured type>
2 //en lugar de simplemente
3 <structured type>
```

Tenga en cuenta que con este tipo de retorno, si, por ejemplo, se va a devolver una instancia local FB del tipo de datos estructurados, se debe usar el operador de referencia **REF=** en lugar del operador de asignación "normal" **:=**.

Las declaraciones y el ejemplo de esta sección se refieren a la llamada de una propiedad. Sin embargo, son igualmente transferibles a otras llamadas que ofrecen valores devueltos (por ejemplo, métodos o funciones).

EJEMPLO:

Declaración de la estructura **ST_Sample** (STRUCTURE):

```
1 TYPE ST_Sample :
2 STRUCT
3     bVar : BOOL;
4     nVar : INT;
5 END_STRUCT
6 END_TYPE
```

Declaración del bloque de funciones **FB_Sample**:

```
1 FUNCTION_BLOCK FB_Sample
2 VAR
3     stLocal : ST_Sample;
4 END_VAR
```

Declaración de la propiedad FB_Sample.MyProp con el tipo de devolución **"REFERENCE TO ST_Sample"**:

```
1 PROPERTY MyProp : REFERENCE TO ST_Sample
```

Implementación del método Get de la propiedad **FB_Sample.MyProp**:

```
1 MyProp REF= stLocal;
```

Implementación del método Set de la propiedad **FB_Sample.MyProp**:

```
1 stLocal := MyProp;
```

Llamando a los métodos Get y Set en el programa principal **MAIN**:

```

1  PROGRAM MAIN
2  VAR
3      fbSample      : FB_Sample;
4      nSingleGet    : INT;
5      stGet         : ST_Sample;
6      bSet          : BOOL;
7      stSet         : ST_Sample;
8  END_VAR
9  // Get - single member and complete structure possible
10 nSingleGet := fbSample.MyProp.nVar;
11 stGet      := fbSample.MyProp;
12
13 // Set - only complete structure possible
14 IF bSet THEN
15     fbSample.MyProp REF= stSet;
16     bSet              := FALSE;
17 END_IF

```

Mediante la declaración del tipo devuelto de la propiedad MyProp como "**REFERENCE TO ST_Sample**" y mediante el uso del operador de referencia **REF=** en el método Get de esta propiedad, se puede acceder a un único elemento del tipo de datos estructurados devuelto directamente al llamar a la propiedad.

```

1  VAR
2      fbSample      : FB_Sample;
3      nSingleGet    : INT;
4  END_VAR
5  nSingleGet := fbSample.MyProp.nVar;

```

Si el tipo de retorno solo se declarara como "ST_Sample", la estructura devuelta por la propiedad tendría que asignarse primero a una instancia de estructura local. Los elementos de estructura individuales podrían consultarse sobre la base de la instancia de estructura local.

```

1  VAR
2      fbSample      : FB_Sample;
3      stGet         : ST_Sample;
4      nSingleGet    : INT;
5  END_VAR
6  stGet             := fbSample.MyProp;
7  nSingleGet := stGet.nVar;

```

RETORNO POR INTERFACE:

Ejemplo de declaración de un método que nos devuelve una variable del tipo **INTERFACE**.

```

1  METHOD Method1 : interface1

```

RETORNO POR FUNCTION_BLOCK:

Ejemplo de declaración de un método que nos devuelve una variable del tipo **FUNCTION_BLOCK**.

```

1  METHOD Method1 : FB1

```

4.4 Objeto Propiedad

Propiedades:

Las propiedades son las principales variables de una clase. Se pueden utilizar como una alternativa a la clase regular o E/S del bloque de funciones. Las propiedades tienen métodos Get "Obtener" y Set "Establecer" que permiten acceder y/o cambiar las variables:

- Get - Método que devuelve el valor de una variable.
- Set - Método que establece el valor de una variable.

Al eliminar el método "Obtener" o "Establecer", un programador puede hacer que las propiedades sean "de solo escritura" o "solo lectura", respectivamente. Dado que estos son métodos, significa que las propiedades pueden:

- Tener sus propias variables internas.
 - Realizar operaciones antes de devolver su valor.
 - No es necesario adjuntar la variable devuelta a una entrada o salida en particular (o variable interna) de la POU, puede devolver un valor basado en una determinada combinación de sus variables.
 - Ser accedido por evento en lugar de ser verificado en cada ciclo de ejecución.
-

Propiedades: Getters & Setters:

para modificar directamente nuestras propiedades lo que se busca es que se haga a través de los metodos Getters y Setters, el cual varía la escritura según el lenguaje pero el concepto es el mismo.

Por lo tanto, una declaración de propiedad tiene la siguiente estructura:

```
1  PROPERTY <Access specifier> <Name> : <Datatype>
```

En el Objeto Propiedad es obligatorio que retorne un valor.

Especificadores de acceso:

Al igual que con los métodos, las propiedades también pueden tomar los siguientes especificadores de acceso: **PUBLIC** , **PRIVATE** , **PROTECTED** , **INTERNAL** y **FINAL** . Cuando no se define ningún especificador de acceso, la propiedad es **PUBLIC** . Además, también se puede especificar un especificador de acceso para cada setter y getter. Esto tiene prioridad sobre el propio especificador de acceso de la propiedad.

Las propiedades son reconocibles por las siguientes características:

Especificador de acceso:

- **PUBLIC:**
- Corresponde a la especificación de modificador sin acceso.
- **PRIVATE:**
- El acceso a la propiedad está limitado solo al bloque de funciones.
- **PROTECTED:**
- El acceso a la propiedad está limitado al programa o al bloque de función y sus derivados.
- **INTERNAL:**
- El acceso a la propiedad está limitado al espacio de nombres, es decir, a la biblioteca.
- **FINAL:**
- No se permite sobrescribir la propiedad en un derivado del bloque de funciones. Esto significa que la propiedad no se puede sobrescribir ni extender en una subclase posiblemente existente.
- Las propiedades pueden ser abstractas, lo que significa que una propiedad no tiene una implementación inicial y que la implementación real se proporciona en el bloque de funciones derivado.

Los pragmas son muy útiles para monitorear propiedades en modo en línea. Para esto, escríbalos en la parte superior de las declaraciones de propiedades (attribute 'monitoring'):

{attribute 'monitoring' := 'variable'}: Al acceder a una propiedad, TwinCAT almacena el valor real en una variable y muestra el valor de esta última. Este valor puede volverse obsoleto si el código ya no accede a la propiedad.

{attribute 'monitoring' := 'call'}: Cada vez que se muestra el valor, TwinCAT llama al código del descriptor de acceso Get. Cualquier efecto secundario, provocado por ese código, puede aparecer en el seguimiento.

Links del Objeto Propiedad:

- [Documentación de Codesys del Objeto propiedad](#)
- [Documentación de Beckhoff del Objeto propiedad](#)
- [utilizing-properties,twincontrols.com](#)
- [object-oriented-programming-in-programmable-logic-controllers-plc-whats-really-new,en.grse.de](#)
- [TC07.Beckhoff TwinCAT3 Function Block-Part2 Property.JP- DUT](#)

4.5 Herencia

4.5.1 Herencia Bloque de Funcion

Herencia Bloque de Funcion:

Los bloques de funciones son un medio excelente para mantener las secciones del programa separadas entre sí. Esto mejora la estructura del software y simplifica significativamente la reutilización. Anteriormente, ampliar la funcionalidad de un bloque de funciones existente siempre era una tarea delicada. Esto significó modificar el código o programar un nuevo bloque de funciones alrededor del bloque existente (es decir, el bloque de funciones existente se incrustó efectivamente dentro de un nuevo bloque de funciones). En el último caso, fue necesario crear todas las variables de entrada nuevamente y asignarlas a las variables de entrada para el bloque de funciones existente. Lo mismo se requería, en sentido contrario, para las variables de salida.

TwinCAT 3 y Codesys (IEC61131-3) introduce el concepto de herencia. La herencia es uno de los principios fundamentales de la programación orientada a objetos. La herencia implica derivar un nuevo bloque de funciones a partir de un bloque de funciones existente. A continuación, se puede ampliar el nuevo bloque. En la medida permitida por los especificadores de acceso del bloque de funciones principal, el nuevo bloque de funciones hereda todas las propiedades y métodos del bloque de funciones principal. Cada bloque de funciones puede tener cualquier número de bloques de funciones secundarios, pero solo un bloque de funciones principal. La derivación de un bloque de funciones se produce en la nueva declaración del bloque de funciones. El nombre del nuevo bloque de funciones va seguido de la palabra clave EXTENDS seguida del nombre del bloque de funciones principal. Por ejemplo:

```
1 FUNCTION_BLOCK PUBLIC FB_NewEngine EXTENDS FB_Engine
```

El nuevo bloque de funciones derivado (FB_NewEngine) posee todas las propiedades y métodos de su padre (FB_Engine). Sin embargo, los métodos y las propiedades solo se heredan cuando el especificador de acceso lo permite.

El bloque de funciones secundario también hereda todas las variables **Locales**, **VAR_INPUT** , **VAR_OUTPUT** y **VAR_IN_OUT** del bloque de funciones principal. Este comportamiento no se puede modificar mediante especificadores de acceso.

Si los métodos o las propiedades del bloque de funciones principal se han declarado como PROTECTED, el bloque de funciones secundario (FB_NewEngine) podrá acceder a ellos, pero no desde fuera de FB_NewEngine .

La herencia se aplica solo a las POU de tipo FUNCTION_BLOCK.

ESPECIFICADORES DE ACCESO:

Las declaraciones FUNCTION_BLOCK , FUNCTION o PROGRAM pueden incluir un especificador de acceso. Esto restringe el acceso y, en su caso, la capacidad de heredar.

- **PUBLIC:**

Cualquiera puede llamar o crear una instancia de la POU. Además, si la POU es un FUNCTION_BLOCK , se puede usar para la herencia. No se aplican restricciones.

- **INTERN:**

La POU solo se puede utilizar dentro de su propio espacio de nombres. Esto permite que las POU estén disponibles solo dentro de una determinada biblioteca, por ejemplo.

- **FINAL:**

El FUNCTION_BLOCK no puede servir como un bloque de funciones principal. Los métodos y las propiedades de esta POU no se pueden heredar. FINAL solo está permitido para POU del tipo FUNCTION_BLOCK .

La configuración predeterminada donde no se define ningún especificador de acceso es PUBLIC. Los especificadores de acceso PRIVATE y PROTECTED no están permitidos en las declaraciones de POU.

Si planea utilizar la herencia, la declaración del bloque de funciones tendrá la siguiente estructura:

```
1 FUNCTION_BLOCK <Access specifier> <Name> EXTENDS <Name basic function block>
```

MÉTODOS DE SOBRESCRITURA:

El nuevo FUNCTION_BLOCK FB_NewEngine , que se deriva de FB_Engine , puede contener propiedades y métodos adicionales. Por ejemplo, podemos agregar la propiedad Gear . Esta propiedad se puede utilizar para consultar y cambiar la marcha actual. Es necesario configurar getters y setters para esta propiedad.

Sin embargo, también debemos asegurarnos de que el parámetro nGear del método Start() se pase a esta propiedad. Debido a que el bloque de funciones principal FB_Engine no tiene acceso a esta nueva propiedad, se debe crear un nuevo método con exactamente los mismos parámetros en FB_NewEngine . Copiamos el código existente al nuevo método y agregamos nuevo código para que el parámetro nGear se pase a la propiedad Gear .

```
1 METHOD PUBLIC Start
2 VAR_INPUT
3   nGear : INT := 2;
4   fVelocity : LREAL := 8.0;
5 END_VAR
6
7 IF (fVelocity < MaxVelocity) THEN
8   velocityInternal := fVelocity;
9 ELSE
10  velocityInternal := MaxVelocity;
11 END_IF
12 Gear := nGear; // new
```

La línea 12 copia el parámetro nGear a la propiedad Gear.

Cuando un método o propiedad que ya está presente en el bloque de funciones principal se redefine dentro del bloque de funciones secundario, esto se denomina sobrescritura. El bloque de funciones FB_NewEngine sobrescribe el método Start() .

Por lo tanto, FB_NewEngine tiene la nueva propiedad Gear y sobrescribe el método Start() .

Herencia

```
1 fbNewEngine.Start(1, 7.5);
```

llama al método Start() en FB_NewEngine, ya que este método ha sido redefinido (sobrescrito) en FB_NewEngine .

Mientras que:

```
1 fbNewEngine.Stop();
```

llama al método Stop() desde FB_Engine . El método Stop() ha sido heredado por FB_NewEngine de FB_Engine .

LINKS:

- [stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](https://stefanhenneken.net/iec-61131-3-methods-properties-and-inheritance)
- [Simple Codesys OOP - Inheritance](#)

4.5.2 Herencia Estructura

Herencia Estructura:

Al igual que los bloques de funciones, las estructuras se pueden ampliar. La estructura obtiene entonces las variables de la estructura básica además de sus propias variables.

Crear una estructura que extienda a otra Estructura:

```
1  TYPE ST_Base1 :
2  STRUCT
3      bBool1: BOOL;
4      iINT : INT;
5      rReal : REAL;
6  END_STRUCT
7  END_TYPE
```

```
1  TYPE ST_Sub1 EXTENDS ST_Base1:
2  STRUCT
3      ttime :TIME;
4      tton  : TON;
5  END_STRUCT
6  END_TYPE
```

```
1  TYPE ST_Sub2 EXTENDS ST_Sub1 :
2  STRUCT
3      bBool2: BOOL; // No se podría llamar la variable bBool1 porque la tenemos declarada en la estructura ST_Base1
4  END_STRUCT
5  END_TYPE
```

```
1  PROGRAM MAIN
2  VAR
3      stestructura1 : ST_Sub1;
4      stestructura2 : ST_Sub2;
5  END_VAR
6
7  //Extensión de Estructura:
8  stestructura1.bBool1;
9  stestructura1.iINT;
10 stestructura1.rReal;
11 stestructura1.ttime;
12 stestructura1.tton(in:= TRUE, pt:=T#1S);
13
14 stestructura2.bBool1;
15 stestructura2.iINT;
16 stestructura2.rReal;
17 stestructura2.ttime;
18 stestructura2.tton(in:= TRUE, pt:=T#1S);
19 stestructura2.bBool2;
```

- De esta forma de extender una Estructura por Herencia no se pueden repetir el mismo nombre de variable declarada con las estructuras extendidas.
- También sin usar EXTENDS para la Estructura podríamos realizarlo de la siguiente forma:

```

1  TYPE ST_2 :
2  STRUCT
3      bBool : BOOL;
4  END_STRUCT
5  END_TYPE

```

```

1  TYPE ST_1:
2  STRUCT
3      sStruct : ST_2;
4      sstring : STRING(80);
5  END_STRUCT
6  END_TYPE

```

```

1  PROGRAM MAIN
2  VAR
3      stestructura11 : ST_1;
4  END_VAR
5
6  stestructura11.sstring;
7  stestructura11.sStruct.bBool; //el resultado es que queda mas anidado

```

- De esta forma si que se pueden declarar el mismo nombre de la variable en diferentes Estructuras, ya que al estar anidadas no existe el problema anterior.
- No se permite la herencia múltiple de la siguiente forma:

```

1  TYPE ST_Sub EXTENDS ST_Base1,ST_Base2 :
2  STRUCT

```

LINKS:

- [infosys.beckhoff.com, Extends Structure](https://infosys.beckhoff.com/Extends-Structure)
- [help.codesys.com, Structure](https://help.codesys.com/Structure)
- [help.codesys.com, Structure](https://help.codesys.com/Structure)
- [help.codesys.com, Structure](https://help.codesys.com/Structure)

4.5.3 Herencia Interface

Herencia Interface:

Al igual que los bloques de funciones, las interfaces se pueden ampliar. A continuación, la interface obtiene los métodos de interface y las propiedades de la interface básica, además de los suyos propios.

Cree una interface que amplíe otra interface mediante la extensión:

```
1  INTERFACE I_Sub1 EXTENDS I_Base1, I_Base2
```

- Se permite la herencia múltiple mediante la extensión de interfaces:

```
1  INTERFACE I_Sub2 EXTENDS I_Sub1
```

- Se permite la herencia múltiple para las interfaces. Es posible que una interfaz amplíe a más de una interface.

LINKS:

- [infosys.beckhoff.com, Extends Interface](https://infosys.beckhoff.com/Extends_Interface)
- [help.codesys.com, Extends Interface](https://help.codesys.com/Extends_Interface)

4.6 THIS puntero

THIS^ puntero:

El puntero **THIS^** se utiliza para referenciar la instancia actual de una clase en un programa orientado a objetos. En otras palabras, cuando se crea un objeto de una clase, el puntero **THIS^** se utiliza para acceder a los atributos y métodos de ese objeto específico. Por ejemplo, si tenemos una clase llamada "Motor" con un atributo "velocidad" y un método "acelerar", al crear un objeto de la clase Motor, podemos utilizar el puntero **THIS^** para hacer referencia a ese objeto y modificar su velocidad o acelerar.

El puntero **THIS^** está disponible para todos los bloques de funciones y apunta a la instancia de bloque de funciones actual. Este puntero es necesario siempre que un método contenga una variable local que oculte una variable en el bloque de funciones.

Una declaración de asignación dentro del método establece el valor de la variable local. Si queremos que el método establezca el valor de la variable local en el bloque de funciones, necesitamos usar el puntero **THIS^** para acceder a él.

Al igual que con el puntero **SUPER**, el puntero **THIS** también debe estar siempre en mayúsculas.

```
1  THIS^.METH_DoIt();
```

Ejemplos:

- La variable del bloque de funciones nVarB se establece aunque nVarB está oculta.

```

1  FUNCTION_BLOCK FB_A
2  VAR_INPUT
3      nVarA: INT;
4  END_VAR
5
6  nVarA := 1;
7
8  FUNCTION_BLOCK FB_B EXTENDS FB_A
9  VAR_INPUT
10     nVarB : INT := 0;
11 END_VAR
12
13 nVarA := 11;
14 nVarB := 2;
15
16 METHOD DoIt : BOOL
17 VAR_INPUT
18 END_VAR
19 VAR
20     nVarB : INT;
21 END_VAR
22
23 nVarB := 22; // Se establece la variable local nVarB.
24 THIS^.nVarB := 222; // La variable del bloque de funciones nVarB se establece aunque nVarB está oculta.
25
26 PROGRAM MAIN
27 VAR
28     fbMyFbB : FB_B;
29 END_VAR
30
31 fbMyFbB(nVarA:=0, nVarB:= 0);
32 fbMyFbB.DoIt();

```

- Una llamada de función necesita la referencia a la propia instancia de FB.

```

1  FUNCTION F_FunA : INT
2  VAR_INPUT
3      fbMyFbA : FB_A;
4  END_VAR
5  ...;
6
7  FUNCTION_BLOCK FB_A
8  VAR_INPUT
9      nVarA: INT;
10 END_VAR
11 ...;
12
13 FUNCTION_BLOCK FB_B EXTENDS FB_A
14 VAR_INPUT
15     nVarB: INT := 0;
16 END_VAR
17
18 nVarA := 11;
19 nVarB := 2;
20
21 METHOD DoIt : BOOL
22 VAR_INPUT
23 END_VAR
24 VAR
25     nVarB: INT;
26 END_VAR
27
28 nVarB := 22; //Se establece la variable local nVarB.
29 F_FunA(fbMyFbA := THIS^); //F_FunA es llamado via THIS^.
30
31 PROGRAM MAIN
32 VAR
33     fbMyFbB: FB_B;
34 END_VAR
35
36 fbMyFbB(nVarA:=0 , nVarB:= 0);
37 fbMyFbB.DoIt();

```

Links THIS^ pointer:

- [THIS puntero Infosys Beckhoff](#)
- [help.codesys.com, THIS](#)

- [stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](https://stefanhenneken.net/iec-61131-3-methods-properties-and-inheritance)

4.7 SUPER puntero

SUPER^ puntero:

En la programación orientada a objetos (OOP) en PLCs, el puntero SUPER^ se utiliza para referirse al objeto o instancia de una clase superior o padre. Supongamos que tienes una clase llamada "Sensor" y otra clase llamada "Sensor_de_Temperatura", que hereda de la primera. La clase "Sensor" es la clase padre o superior y la clase "Sensor_de_Temperatura" es la clase hija o inferior. Si estás programando en la clase "Sensor_de_Temperatura" y necesitas acceder a un método o propiedad de la clase "Sensor", puedes utilizar el puntero SUPER^ para referirte a la instancia de la clase "Sensor" a la que pertenece el objeto actual. Por ejemplo, si quieres acceder al método "obtener_valor()" de la clase "Sensor", puedes hacerlo así: SUPER^.obtener_valor(). Esto indica que quieres llamar al método "obtener_valor()" de la instancia de la clase "Sensor" a la que pertenece el objeto actual.

cada bloque de funciones que se deriva de otro bloque de funciones tiene acceso a un puntero llamado SUPER^. Esto se puede usar para acceder a elementos (métodos, propiedades, variables locales, etc.) desde el bloque de funciones principal.

En lugar de copiar el código del bloque de funciones principal al nuevo método, el puntero SUPER^ se puede usar para llamar al método desde el bloque de funciones. Esto elimina la necesidad de copiar el código.

```
1  SUPER^();           // Llamada del cuerpo FB de la clase base.
2  SUPER^.METH_DoIt(); // Llamada del método METH_DoIt que se implementa en la clase base.
```

Ejemplo:

- Usando los punteros SUPER y THIS:

Bloque de Función -- FB_Base:

```
1  FUNCTION_BLOCK FB_Base
2  VAR_OUTPUT
3      nCnt : INT;
4  END_VAR
```

Metodo -- FB_Base.METH_DoIt:

```
1  METHOD METH_DoIt : BOOL
2      nCnt := -1;
```

Metodo -- FB_Base.METH_DoAlso:

```
1  METHOD METH_DoAlso : BOOL
2      METH_DoAlso := TRUE;
```

Bloque de Función -- FB_1:

```
1  FUNCTION_BLOCK FB_1 EXTENDS FB_Base
2  VAR_OUTPUT
3      nBase: INT;
4  END_VAR
5  THIS^.METH_DoIt(); // llamada al metodo METH_DoIt del FB_1.
6  THIS^.METH_DoAlso();
7
8  SUPER^.METH_DoIt(); // llamada al metodo METH_DoIt del FB_Base.
9  SUPER^.METH_DoAlso();
10 nBase := SUPER^.nCnt;
```

Metodo -- FB_1.METH_DoIt:

```
1  METHOD METH_DoIt : BOOL
2      nCnt := 1111;
3      METH_DoIt := TRUE;
```

Metodo -- FB_1.METH_DoAlso:

```
1  METHOD METH_DoAlso : BOOL
2  nCnt := 123;
3  METH_DoAlso := FALSE;
```

Programa MAIN:

```
1  PROGRAM MAIN
2  VAR
3      fbMyBase : FB_Base;
4      fbMyFB_1 : FB_1;
5      nTHIS    : INT;
6      nBase     : INT;
7  END_VAR
8  fbMyBase();
9  nBase := fbmyBase.nCnt;
10 fbMyFB_1();
11 nTHIS := fbMyFB_1.nCnt;
```

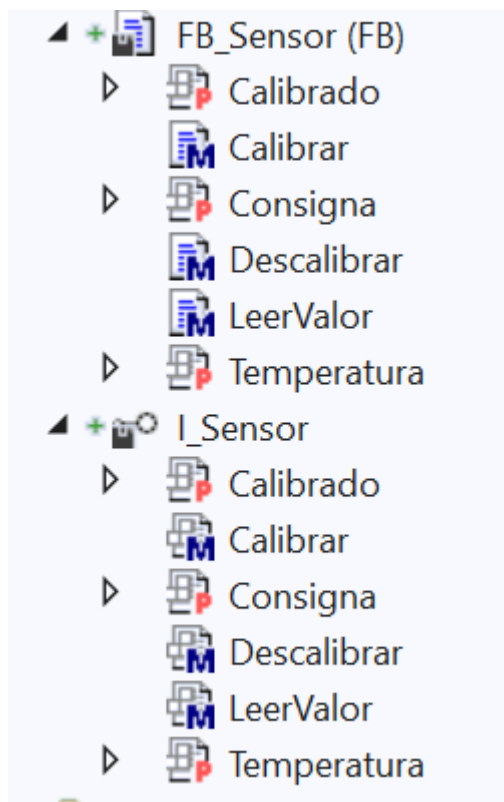
Links SUPER^ pointer:

- [🔗 SUPER puntero Infosys Beckhoff](#)
- [🔗 help.codesys.com, SUPER](#)
- [🔗 stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance](#)

4.8 Interface

Interface:

En la programación orientada a objetos (OOP) en PLCs, una interfaz es un tipo de estructura que define un conjunto de métodos y propiedades que una clase debe implementar. En otras palabras, una interfaz define un contrato entre diferentes partes del código para asegurar que se cumplan ciertos requisitos y se mantenga una estructura coherente. En términos prácticos, esto significa que cuando se crea una clase que implementa una interfaz, esa clase debe proporcionar los métodos y propiedades definidos en la interfaz. Esto permite que diferentes clases compartan un conjunto común de métodos y propiedades y se comuniquen entre sí de manera coherente. Por ejemplo, si tienes una interfaz **"I_Sensor"** con los **métodos:** **"LeerValor"**, **"Calibrar"** y **"Descalibrar"** y las **Propiedades:** **"Temperatura"**, **"Consigna"** y **"Calibrado"** cualquier clase que implemente esa interfaz debe proporcionar esos tres métodos y las tres propiedades. Esto asegura que cualquier otra parte del código que trabaje con esa clase pueda confiar en que esos métodos y propiedades estarán disponibles.



- Una interfaz es una clase que contiene métodos y propiedades sin implementación.
- La interfaz se puede implementar en cualquier clase, pero esa clase debe implementar todos sus métodos. y propiedades.
- Si bien la herencia es una relación "es un", las interfaces se pueden describir como "se comporta como" o "tiene una" relación.
- Las interfaces son objetos que permiten que varias clases diferentes tengan algo en común con menos dependencias. Las clases y los bloques de funciones pueden implementar varias interfaces diferentes. Uno puede pensar en los métodos y propiedades de la interfaz como acciones que significan cosas diferentes dependiendo de quién los esté ejecutando. Por ejemplo, la palabra "Correr" significa "mover a una velocidad más rápido que un paseo" para un ser humano, pero significa "ejecutar" para las computadoras.
- Las clases o bloques de funciones que no comparten similitudes pueden implementar la misma interfaz. En este caso, la implementación de los métodos en cada clase puede ser totalmente diferente. Esto abre muchos enfoques de programación poderosos:
- Las POU pueden llamar a una interfaz para ejecutar un método o acceder a una propiedad, sin saber cuál clase o FB con el que se trata o cómo va a ejecutar la operación. La interfaz luego apunta a una clase o bloque de función que implementa la interfaz y la operación que es ejecutado.
- Los programadores pueden crear cajas de interruptores fácilmente personalizables usando polimorfismo.

Links Interface:

- [Codesys Comando 'Implementar interfaces'](#)
- [Codesys Objeto Interface](#)
- [Codesys Implementando Interfaces](#)
- [Beckhoff Objeto Interface](#)
- [Beckhoff Implementando Interfaces](#)
- [Extender Interfaces, Infosys Beckhoff](#)
- [TC09.Beckhoff TwinCAT3 Function Block-Part4 Interface.JP](#)

4.9 puntero y referencia

Puntero y Referencia:

. ¿Qué es un puntero?

- Es un dato que apunta o señala hacia una dirección de memoria.
- Es una variable que contiene la dirección de memoria donde “vive” la variable.
- Con el empleo de punteros se accede a la memoria de forma directa, por lo que es una buena técnica para reducir el tiempo de ejecución de un programa y otras muchas más funcionalidades.

. Tipos de Punteros:

- Hay un tipo de puntero para cada tipo de dato, programa, Function Block, funciones, etc.
- Según sea el “objeto” al que se desea acceder se necesita un puntero de un tipo u otro.

. Declaración de punteros:

El compilador necesita conocer todos los punteros que se vayan a emplear en el proyecto, por lo que hay que declararlos, como cualquier otra variable. En el código se muestra el script necesario para la declaración de varios tipos de punteros:

```

1  // Un puntero no deja de ser una variable, la diferencia está en que su contenido no es un valor determinado sino que es la dirección
2  // de memoria donde se ubica la variable de la que se quiere leer o escribir su valor. Y al igual que hay que declarar todas las
3  // variables del tipo correspondiente. también hay que declarar todas las variables -punteros- que contendrán esas direcciones de
4  // memoria y su correspondiente tipo.
5  VAR
6      stTest1 : stTipo1; //Declara una estructura de datos del tipo stTipo1.
7
8      pin01 : POINTER TO INT; //Declara un puntero para acceder a variables del tipo INT.
9      ps20  : POINTER TO STRING[20]; //Declara un puntero para acceder a variables del tipo STRING de 20 caracteres.
10     pa20  : POINTER TO ARRAY [1..20] OF INT; //Declara un puntero para acceder a variables del tipo ARRAY de 20 elementos del tipo INT.
11     pDword : POINTER TO DWORD; //Declara un puntero para acceder a variables del tipo DWORD.
12     past1  : POINTER TO stTipo1; //Declara un puntero para acceder a variables del tipo stTipo1.
13     pReal  : POINTER TO REAL; //Declara un puntero para acceder a variables del tipo REAL.
14 END_VAR

```

. Como saber qué dirección asignar al puntero:

- Para poder acceder a una variable mediante un puntero se necesita conocer su dirección de memoria, Para ello se dispone de un operador llamado **ADR** que asigna la dirección de la variable deseada, al puntero.
- Es conveniente verificar que el valor del puntero no es cero, antes de utilizarlo. Por otra parte, para poder leer / escribir el valor de la variable, a la que señala el puntero, se dispone del operador de contenido **^**. Cuando se hace referencia al

contenido, de la dirección de memoria apuntada, se habla de desreferenciar el puntero. En el siguiente código se muestra un ejemplo:

```

1  PROGRAM SR_Main_02
2  VAR
3      in01      : INT; //Declaración de la variable in01 de tipo entero.
4      in02      : INT := 123; //Declaración e inicialización de la variable in02 de tipo entero.
5      in03      : INT; //Declaración de la variable in03 de tipo entero.
6
7      pint      : POINTER TO INT; //Declaración de un puntero para acceder a variables del tipo entero.
8  END_VAR
9
10 // Ejemplo de uso básico de los operadores ADR y del operador de contenido ^
11 // Se muestra como asignar a un puntero la dirección de memoria de una variable y como leer/escribir
12 // así como un ejemplo de acceso a variables locales de otros programas.
13
14 pint := ADR(in01); //Asignamos al puntero la dirección de memoria donde se ubica la variable in01.
15 pint^ := 44;       //A la posición de memoria indicada por el puntero, le asignamos el valor 44
16                    //Por tanto a la variable in01 se le ha escrito el valor 44.
17
18 in02 := in01; // in02 será igual a 44.
19
20 pint := ADR(in02); //Cambiamos la dirección para acceder a la dirección de la variable in02.
21 in03 := pint^; // in03 tomara el valor del contenido de la posición de memoria contenida en el
22            // que hemos asignado la dirección de in02, por tanto in03= 123.
23
24 pint := ADR(SR_Main_01.inLocalAway); //Cargamos la dirección de memoria de una variable local de
25            // otro programa, la que sería inaccesible por otros medios.
26 pint^ := 240 ; // La variable local del programa SR_Main_01.inLocalAway tomará el valor 240

```

¿Qué es un acceso indirecto?

Lo primero, decir que no tiene nada que ver con un puntero. Un acceso indirecto permite elegir un número de elemento dentro de un array, hay una variable, llamada índice, que contiene el número del elemento del array al que se desea acceder. En este caso no se puede acceder a ninguna otra variable más allá de los elementos del array, insisto en que no tiene nada que ver con los punteros. Con un puntero se puede acceder a cualquier dato u objeto que esté en la memoria del control. Con un acceso indirecto solo se puede acceder a los elementos de un array. En el siguiente código se muestra unos ejemplos de acceso indirecto a un array:

```

1  PROGRAM SR_Main_01
2  VAR
3      aR20: ARRAY[1..20] OF REAL; //Declara un array de 20 elementos del tipo REAL.
4      inIndex: INT; //Declara la variable de indice del array para el acceso indirecto
5      xNewVal: BOOL; //Indica que hay una nueva lectura del sensor de fuerza.
6      rFuerza: REAL; //Valor de fuerza del sensor.
7  END_VAR
8
9 // Ejemplo01: Se asigna valores del 1 al 20 a cada elemento del array mediante un bucle.
10 FOR inIndex:=1 TO 20 BY 1 DO // Se empieza por el valor de la variable indice a 1, hasta 20
11     aR20[inIndex] := inIndex; // Al elemento aR20[inIndex] se le asigna el valor de inIndex
12 END_FOR; // Se incrementa inIndex y se repite el proceso.
13
14 // Ejemplo02: Creamos un FIFO en el que guardamos un valor analógico de fuera a cada impulso de la señal xNewVal.
15 IF xNewVal THEN // Si hay un nuevo valor de fuerza realizamos el código.
16     xNewVal := FALSE; // Reset de la señal xNewVal.
17
18     FOR inIndex:=20 TO 2 BY -1 DO // Variable indice a 20, hasta 2.
19         aR20[inIndex] := aR20[inIndex-1]; //Desplazamiento de los valores en el FIFO -->
20     END_FOR; // Se decrementa inIndex y se repite el proceso.
21
22     aR20[1] := rFuerza; // Entrada del valor de fuerza en el primer elemento del FIFO.
23 END_IF

```

A este mismo array se puede acceder empleando un puntero, como se verá más adelante, lo que resulta más rápido en tiempo de ejecución, pero no tan claro para quien no suele usar los punteros.

. Acceso a una estructura de datos mediante punteros:

El proceso es el mismo que ya se ha visto para acceder a una variable del tipo INT, pero se tendrá que declarar un puntero del tipo adecuado, que coincida con el tipo de estructura a la que se desea acceder, veámoslo en el siguiente código:

```

1  PROGRAM SR_Main_03
2  VAR
3      stMotor_01 : stMotorCtrl; // Estructura de control del motor 1
4      stMotor_02 : stMotorCtrl; // Estructura de control del motor 2
5      stMotor_03 : stMotorCtrl; // Estructura de control del motor 3
6      pstMotorCtrl : POINTER TO stMotorCtrl; // Puntero para acceder a estructuras del tipo stMotorCtrl.
7      xMarcha : BOOL; // Pulsador marcha motores.
8  END_VAR
9
10 // Ejemplo básico de como acceder a estructuras de datos mediante punteros.
11 // La estructura de datos empleada es una llamada a stMotorCtrl, que coincide un bit de marcha, otro de paro y
12 // valores de velocidad en Rpm y tiempo de aceleración/deceleración.
13
14 // Asignamos valores a la estructura para el control del motor 1.
15
16 stMotor_01.rTpoAcelDecel := 5.4; // Tiempo para acelerar/decelerar hasta alcanzar la velocidad.
17 stMotor_01.rVelRpm := 1436.2; // Velocidad en RPM.
18 stMotor_01.xMotorOff := TRUE; // Bit de paro ON.
19 stMotor_01.xMotorOn := FALSE; // Bit de marcha OFF.
20
21 pstMotorCtrl := ADR(stMotor_01); // Cargamos la dirección de memoria de la estructura del motor 1
22 stMotor_02 := pstMotorCtrl^; // Copia el contenido de la zona de memoria apuntada a la
23 // estructura del motor 2, en este caso el resultado es el mismo
24 // que se obtendría con stMotor_02:= stMotor_01;
25
26 stMotor_03 := stMotor_02; // Copia los mismos valores al motor 3;
27
28 IF xMarcha THEN // Si se pulsa marcha máquina
29     pstMotorCtrl^.xMotorOn := TRUE; // Se activa el bit de marcha al que apunta el puntero (stMotor_01).
30     pstMotorCtrl^.xMotorOff := FALSE; // Se desactiva el bit de paro al que apunta el puntero (stMotor_01)
31 END_IF

```

. Acceso a un array mediante punteros:

El proceso es el mismo que ya se ha visto para acceder a una variable del tipo INT, pero se tendrá que declarar un puntero a un array del número de elementos y tipo de datos adecuados, veámoslo en el siguiente código:

```

1  PROGRAM SR_Main_03
2  VAR
3      aintFIFO : ARRAY[1..20] OF INT; // Array de 20 enteros.
4      aintFIFO2 : ARRAY[1..20] OF INT; // Array de 20 enteros.
5      paint : POINTER TO ARRAY[1..20] OF INT; // Puntero al array.
6      pint : POINTER TO INT; // Puntero a un entero.
7  END_VAR
8
9 // Ejemplo basico de como acceder a arrays mediante punteros:
10 paint := ADR(aintFIFO); // _Asignamos la dirección del array al puntero.
11 paint^[3] := 4; // Dentro del array podemos acceder a un elemento en concreto
12 aintFIFO2 := paint^; // 0 copiar el array apuntado entero, sobre otro array
13 pint := paint + (4 * SIZEOF (INT)); // Tambien se puede crear un puntero a un INT para acceder a uno de los
14 // elementos del array. Tomamos la dirección inicial del array y le
15 // sumamos un offset de tantos bytes como se necesitan para el tipo de
16 // datos INT y lo multiplicamos por el índice del array al que queremos
17 // acceder. SIZEOF (TYPE) retorna el número de bytes según el tipo de datos.
18
19 pint^ := 5; // Asignamos el valor de 5, aintFIFO[5]:=5 sería lo mismo.

```

. Acceso a datos por referencias:

El acceso por referencia no deja de ser un acceso por puntero, pero en este caso la dirección de una referencia es la misma que la del objeto al que apunta. Un puntero tiene su propia dirección y esta contiene la dirección del objeto al que se quiere hacer referencia. Las referencias se inicializan al principio del programa y no pueden cambiar durante su ejecución. A un puntero se le puede cambiar su dirección tanto como sea necesario durante la ejecución del programa. Otra forma de entender las referencias es como si fuesen otra manera de referirse a un mismo objeto/variable, como si fuese un alias. Frente a los punteros, las referencias presentan las siguientes ventajas: 1) Facilidad de uso. 2) Sintaxis más sencilla a la hora de pasar parámetros a funciones. 3) Minimiza errores en la escritura del código. El resumen de todo esto, que se puede prestar a mucha confusión, es que, como se verá más adelante, el gran valor de las referencias es a la hora de pasar grandes cantidades de datos como parámetros de entrada a funciones.

. Diversas formas de pase de parámetros a funciones:

Normalmente una función realiza unas operaciones con unos parámetros de entrada y retorna un valor - o varios - como resultado. En el ejemplo que veremos seguidamente se trata de una función para calcular el área de un rectángulo, a la que le pasaremos los valores del lado A y el lado B para que nos retorne el resultado del área. Lo primero definiremos un tipo de dato [stRectángulo] que contendrá el lado A, el B y el área. Crearemos tres rectángulos, [stRectangulo01], [stRectangulo02] y [stRectangulo03]. Junto con tres variantes de la función para el cálculo del área, [Fc_AreaCalcVal] - pase por valores -, [Fc_AreaCalcPoint] - pase por puntero - [Fc_AreaCalcRef] - pase por referencia - A continuación, el código de las tres funciones:

Pase de valores:

```

1 // Función para calcular el area de un Rectangulo, pasando los valores de los lados del Rectangulo
2 // la función retorna el resultado del area calculado
3
4 FUNCTION Fc_AreaCalcVal : REAL // La función retorna un número real
5 VAR_INPUT
6     i_rASide    : REAL; // Parámetro de entrada que contiene el lado A del rectangulo.
7     i_rBSide    : REAL; // Parámetro de entrada que contiene el lado B del rectangulo.
8 END_VAR
9
10 Fc_AreaCalcVal := i_rASide * i_rBSide; // Retorna el resultado de multiplicar el lado A por el lado B.
```

Pase por puntero:

```

1 // Función para calcular el area de un Rectangulo, con los valores contenidos en una estructura de datos del tipo stRectangulo
2 // La estructura se pasa mediante un puntero a la estructura stRectangulo deseada y la función retorna el resultado a la
3 // misma estructura.
4
5 FUNCTION Fc_AreaCalcPoint
6 VAR_INPUT
7     i_ptstRect : POINTER TO st_Rectangulo; // Puntero de entrada con la dirección de la estructura.
8 END_VAR
9
10 // El valor del area, de la estructura indicada por la dirección del puntero es igual al
11 // valor del lado A de la estructura indicada por la dirección del puntero por
12 // el valor del lado B de la estructura indicada por la dirección del puntero
13 i_ptstRect^.rArea := i_ptstRect^.rASide * i_ptstRect^.rBSide;
```

Pase por Referencia:

```

1 // Función para calcular el area de un Rectangulo, con los valores contenidos en una estructura de datos del tipo stRectangulo
2 // La estructura se pasa por referencia.
3
4 FUNCTION Fc_AreaCalcRef : REAL
5 VAR_INPUT
6     i_Ref : REFERENCE TO stRectangulo
7 END_VAR
8
9 i_Ref.rArea := i_Ref.rASide * i_Ref.rBSide;
```

Ejemplo de código de llamadas a las funciones:

```

1  PROGRAM SR_Main_01
2  VAR
3      inLocalAway : INT; // Variable integer local de SR_Main_01 para ser accedida externamente
4      stRectangulo1 : stRectangulo; // Estructura que contiene los datos del rectangulo1 A, B y su area
5      stRectangulo2 : stRectangulo; // Estructura que contiene los datos del rectangulo2 A, B y su area
6      stRectangulo3 : stRectangulo; // Estructura que contiene los datos del rectangulo3 A, B y su area
7
8      refRectangulo : REFERENCE TO stRectangulo := stRectangulo3; // Hace Referencia a stRectangulo3
9  END_VAR
10
11 // Asignación de valores a los lados de los tres rectángulos.
12
13 // Asignación de valores de los lados del rectángulo 1
14 stRectangulo1.rAside := 44; //Valor del lado A.
15 stRectangulo1.rBside := 32; //Valor del lado B.
16
17 // Asignación de valores de los lados del rectángulo 2
18 stRectangulo2.rAside := 12.8; //Valor del lado A.
19 stRectangulo2.rBside := 320.4; //Valor del lado B.
20
21 // Asignación de valores de los lados del rectángulo 3
22 stRectangulo3.rAside := 1024.2; //Valor del lado A.
23 stRectangulo3.rBside := 2048.4; //Valor del lado B.
24
25 // Cálculo del área del rectángulo pasando valores a la función
26 stRectangulo1.rArea := Fc_AreaCalcVal(i_rAside:=stRectangulo1.rAside, i_rBside:= stRectangulo1.rBside);
27
28 // Cálculo del área del rectángulo pasando un puntero a la función
29 Fc_AreaCalcPoint(ADR(stRectangulo2));
30
31 // Cálculo del área del rectángulo pasando una referencia a la función
32 Fc_AreaCalcRef(refRectangulo);

```

En este caso puede las diferencias pueden parecer insignificantes, puesto que la cantidad de datos que se le pasan a la función son pocos. Pero seguidamente veremos un ejemplo con mayor número de parámetros de entrada para poder apreciar las ventajas del pase de parámetros por, especialmente, referencia y también por puntero.

.Caso de pase de grandes cantidades de datos a funciones:

Cuando se precisa pasar estructuras con gran cantidad de datos a funciones ó a FB's, el pase de parámetros por valores no es el método más adecuado puesto que se requieren gran cantidad de parámetros de entrada, cada parámetro implica crear una nueva variable local de la función, o del FB, lo que supone gasto de memoria y tiempo de ejecución en copiar los datos. Caso de estructuras de datos de varios Kbytes, o arrays de centenares o miles de elementos, este método es impensable. En el caso de tener que pasar grandes cantidades de datos, la solución es el empleo de punteros, o mejor aún, el pase de datos por referencia. Seguidamente se muestra un ejemplo de una función para calcular el valor promedio de un array de 20 elementos, pasando los valores a la función y pasando los valores mediante una referencia.

Código de la función Fc_AverageValues para pase de valores:

!!!!!!!!!!!! VOY POR AQUI!!!!!!!!!!!!

-
- Un puntero de tipo T apunta a un objeto de tipo T (T = tipo de datos básico o definido por el usuario)
 - Un puntero contiene la dirección del objeto al que apunta.
 - La operación fundamental con un puntero se llama "desreferenciar". La desreferenciación en CODESYS se realiza con el símbolo "^"
 - Un puntero puede apuntar a un objeto diferente en un momento diferente.
 - Antes de desreferenciar un puntero y asignarle un valor, siempre debe verificar si un puntero apunta a un objeto. (puntero = 0)?
 - Una referencia del tipo T "apunta" a un objeto del tipo T (T = tipo de datos básico o definido por el usuario).
 - Una referencia debe ser inicializada con un objeto y su "apuntando" a este objeto a través del programa.
 - Una referencia no debe ser desreferenciada como un puntero y puede usarse con la misma sintaxis que el objeto.
 - Otra palabra de referencia es "Alias" (otro nombre) un seudónimo para el objeto.
 - La referencia no tiene dirección propia y un puntero sí. La dirección de la referencia es la misma que la del objeto "puntiagudo".
 - No hay referencia 0, por lo que nunca debe llamar a la referencia si no está inicializada.
 - Debe verificar si tiene una referencia válida con la palabra clave integrada CODESYS "__ISVALIDREF".




El mejor uso de punteros y referencias es cuando desea pasar o devolver un objeto de algún tipo a una función o bloque de funciones por "referencia" porque el objeto es demasiado grande o desea manipular el objeto pasado dentro de la función/bloque de función. Asegúrese de que el lector de su código sepa que va a cambiar el valor del objeto dentro de la función/bloque de funciones si esto es lo que pretende hacer cuando lo pasa como argumento.

.Resumen / Conclusiones:

- La memoria contiene miles y hasta millones de celdas o byte, en las que se ubica el código del programa y todos los datos/variables. Cada celda tiene su número, al que se llama dirección de memoria y que se suele expresar en hexadecimal 16#FA1204 -como ejemplo-
 - Un puntero es una variable, que en lugar de contener un valor contiene una dirección de memoria, en la que "vive" la variable a la que realmente queremos acceder.
 - Al igual que cualquier otra variable, hay que declarar los punteros para que el compilador pueda ubicarlos en la memoria. Recordemos que un puntero es una variable, pero que su contenido es una dirección de memoria.
 - Para cada tipo de variable se precisa el correspondiente tipo de puntero. No se puede acceder a una variable INT con un puntero pensado para acceder a una estructura de datos.
 - Nada tiene que ver el acceso indirecto a un array mediante una variable de índice, con un puntero. En este caso el acceso está limitado al propio array, con el puntero se puede acceder a cualquier dirección de memoria.
 - Con punteros se puede acceder a todo tipo de datos, en una simple línea de código se puede copiar una estructura entera de varios Kbytes de datos. Lo que resulta mucho más rápido.
 - Una referencia se parece mucho a un puntero, para simplificar podríamos decir que es un "alias" de un objeto y que es algo menos crítico que los punteros, su principal utilidad es la de pasar gran cantidad de parámetros a funciones, de forma muy simple y rápida.
 - El pase de parámetros a una función se puede realizar de diversas formas, por valores, por punteros o por referencia, el programador deberá elegir el más adecuado para cada aplicación.
 - Cuando se trata de grandes cantidades de datos el pase de parámetros por referencia o por punteros, serán los adecuados
-

Links de Puntero y Referencia:

- [Perre Garriga,Pointer&Reference](#)
- [Control and use of Pointers In Codesys](#)

-  [help.codesys.com, Pointers](https://help.codesys.com/Pointers)
-  [AT&U, CODESYS - Difference between pointer and reference](#)
-  [AT&U, CODESYS -Difference between pass by vale and pass by Reference](#)

4.10 FB abstracto frente a interfaz

FB Abstracto frente a Interface:

FB Abstracto frente a Interface:

[FB abastracto frente a interfaz, stefanhenneken.net](https://stefanhenneken.net)

5. Tabla de Modificadores de acceso

Modificadores de acceso	FUNCTION_BLOCK - FB	METODO	PROPIEDAD
PUBLIC	Si	Si	Si
INTERNAL	Si	Si	Si
FINAL	Si	Si	Si
ABSTRACT	Si	Si	Si
PRIVATE	No	Si	Si
PROTECTED	No	Si	Si

6. Tipos de variables y variables especiales

6.0.1 Variable types and special variables:

The variable type defines how and where you can use the variable. The variable type is defined during the variable declaration.

6.0.2 Further Information:

- [Local Variables - VAR](#)

- Las variables locales se declaran en la parte de declaración de los objetos de programación entre las palabras clave VAR y END_VAR.
- Puede extender las variables locales con una palabra clave de atributo.
- Puede acceder a variables locales para leer desde fuera de los objetos de programación a través de la ruta de instancia. El acceso para escribir desde fuera del objeto de programación no es posible; Esto será mostrado por el compilador como un error.
- Para mantener la encapsulación de datos prevista, se recomienda encarecidamente no acceder a las variables locales de un POU desde fuera del POU, ni en modo de lectura ni en modo de escritura. (Otros compiladores de lenguaje de alto nivel también generan operaciones de acceso de lectura a variables locales como errores). Además, con los bloques de funciones de biblioteca no se puede garantizar que las variables locales de un bloque de funciones permanezcan sin cambios durante las actualizaciones posteriores. Esto significa que es posible que el proyecto de aplicación ya no se pueda compilar correctamente después de la actualización de la biblioteca.
- También observe aquí la regla SA0102 del Análisis Estático, que determina el acceso a las variables locales para la lectura desde el exterior.

- [Input Variables - VAR_INPUT](#)

- Las variables de entrada son variables de entrada para un bloque de funciones.
- VAR_INPUT variables se declaran en la parte de declaración de los objetos de programación entre las palabras clave VAR_INPUT y END_VAR.
- Puede ampliar las variables de entrada con una palabra clave de atributo.

- [Output Variables - VAR_OUTPUT](#)

- Las variables de salida son variables de salida de un bloque de funciones.
- VAR_OUTPUT variables se declaran en la parte de declaración de los objetos de programación entre las palabras clave VAR_OUTPUT y END_VAR. TwinCAT devuelve los valores de estas variables al bloque de función de llamada. Allí puede consultar los valores y continuar usándolos.
- Puede ampliar las variables de salida con una palabra clave de atributo.

- [Input/Output Variables - VAR_IN_OUT, VAR_IN_OUT CONSTANT](#)

- [Global Variables - VAR_GLOBAL](#)

- Solo es posible su declaración en GVL (Lista de Variables Global)

- [Temporary Variable - VAR_TEMP](#)

- Esta funcionalidad es una extensión con respecto a la norma IEC 61131-3.
- Las variables temporales se declaran localmente entre las palabras clave VAR_TEMP y END_VAR.
- VAR_TEMP declaraciones sólo son posibles en **programas y bloques de funciones**.
- TwinCAT reinicializa las variables temporales cada vez que se llama al bloque de funciones.
- La aplicación sólo puede acceder a variables temporales en la parte de implementación de un programa o bloque de funciones.

- [Static Variables - VAR_STAT](#)

- Esta funcionalidad es una extensión con respecto a la norma IEC 61131-3.
- Las variables estáticas se declaran localmente entre las palabras clave VAR_STAT y END_VAR. TwinCAT inicializa las variables estáticas cuando se llama por primera vez al bloque de funciones respectivo.
- Puede tener acceso a las variables estáticas sólo dentro del espacio de nombres donde se declaran las variables (como es el caso de las variables estáticas en C). Sin embargo, las variables estáticas conservan su valor cuando la aplicación sale del bloque de funciones. Puede utilizar variables estáticas, como contadores para llamadas a funciones, por ejemplo.
- Puede extender variables estáticas con una palabra clave de atributo.
- Las variables estáticas solo existen una vez. Esto también se aplica a las variables estáticas de un bloque de funciones o un método de bloque de funciones, incluso si el bloque de funciones se instancia varias veces.

- [External Variables - VAR_EXTERNAL](#)

- Las variables externas son variables globales que se "importan" en un bloque de funciones.
- Puede declarar las variables entre las palabras clave VAR_EXTERNAL y END_VAR. Si la variable global no existe, se emite un mensaje de error.
- En TwinCAT 3 PLC no es necesario que las variables se declaren como externas. La palabra clave existe para mantener la compatibilidad con IEC 61131-3.
- Si, no obstante, utiliza variables externas, asegúrese de abordar las variables asignadas (con AT %I o AT %Q) sólo en la lista global de variables. El direccionamiento adicional de las instancias de variables locales daría lugar a duplicaciones en la imagen del proceso.
- Estas variables declaradas también tiene que estar declarada la misma variable con el mismo nombre en una GVL (Lista de Variables Global)

- [Instance Variables - VAR_INST](#)

- TwinCAT crea una variable VAR_INST de un método no en la pila de métodos como las variables VAR, sino en la pila de la instancia del bloque de funciones. Esto significa que la variable VAR_INST se comporta como otras variables de la instancia del bloque de función y no se reinicializa cada vez que se llama al método.
- VAR_INST variables solo están permitidas en los métodos de un bloque de funciones, y el acceso a dicha variable solo está disponible dentro del método. Puede supervisar los valores de las variables de instancia en la parte de declaración del método.
- Las variables de instancia no se pueden extender con una palabra clave de atributo.
- [Remanent Variables - PERSISTENT, RETAIN](#) Las variables remanentes pueden conservar sus valores más allá del tiempo de ejecución habitual del programa. Las variables remanentes se pueden declarar como variables RETAIN o incluso más estrictamente como variables PERSISTENTES en el proyecto PLC.

Un requisito previo para la funcionalidad completa de las variables RETAIN es un área de memoria correspondiente en el controlador (NovRam). Las variables persistentes se escriben solo cuando TwinCAT se apaga. Esto requerirá generalmente un UPS correspondiente. Excepción: Las variables persistentes también se pueden escribir con el bloque de función FB_WritePersistentData.

Si el área de memoria correspondiente no existe, los valores de las variables RETAIN y PERSISTENT se pierden durante un corte de energía.

La declaración AT no debe utilizarse en combinación con VAR RETAIN o VAR PERSISTENT.

6.0.3 Variables persistentes

Puede declarar variables persistentes agregando la palabra clave PERSISTENT después de la palabra clave para el tipo de variable (VAR, VAR_GLOBAL, etc.) en la parte de declaración de los objetos de programación.

Las variables PERSISTENTES conservan su valor después de una terminación no controlada, un Reset cold o una nueva descarga del proyecto PLC. Cuando el programa se reinicia, el sistema continúa funcionando con los valores almacenados. En este caso, TwinCAT reinicializa las variables "normales" con sus valores iniciales especificados explícitamente o con las inicializaciones predeterminadas. En otras palabras, TwinCAT solo reinicializa las variables PERSISTENTES durante un origen de Restablecer.

Un ejemplo de aplicación para variables persistentes es un contador de horas de funcionamiento, que debe continuar contando después de un corte de energía y cuando el proyecto PLC se descarga nuevamente.

Evite usar el tipo de datos POINTER TO en listas de variables persistentes, ya que los valores de dirección pueden cambiar cuando el proyecto PLC se descargue nuevamente. TwinCAT emite las advertencias correspondientes del compilador. Declarar una variable local como PERSISTENTE en una función no tiene ningún efecto. La persistencia de datos no se puede utilizar de esta manera. El comportamiento durante un restablecimiento en frío puede verse influenciado por el pragma 'TcInitOnReset'

6.0.4 Variables RETAIN

Puede declarar variables RETAIN agregando la palabra clave RETAIN después de la palabra clave para el tipo de variable (VAR, VAR_GLOBAL, etc.) en la parte de declaración de los objetos de programación.

Las variables declaradas como RETAIN dependen del sistema de destino, pero normalmente se administran en un área de memoria separada que debe protegerse contra fallas de energía. El llamado controlador Retain asegura que las variables RETAIN se escriban al final de un ciclo PLC y solo en el área correspondiente de la NovRam. El manejo del controlador de retención se describe en el capítulo "Conservar datos" de la documentación de C/C++.

Las variables RETAIN conservan su valor después de una terminación incontrolada (corte de energía). Cuando el programa se reinicia, el sistema continúa funcionando con los valores almacenados. En este caso, TwinCAT reinicializa las variables "normales" con sus valores iniciales especificados explícitamente o con las inicializaciones predeterminadas. TwinCAT reinicializa las variables RETAIN en un origen de restablecimiento.

Una posible aplicación es un contador de piezas en una planta de producción, que debe seguir contando después de un corte de energía.

Si declara una variable local como RETAIN en un programa o bloque de funciones, TwinCAT almacena esta variable específica en el área de retención (como una variable RETAIN global). Si declara una variable local en una función como RETAIN, esto no tiene efecto. TwinCAT no almacena la variable en el área de retención.

6.0.5 Cuadro general completo

El grado de retención de las variables RETAIN se incluye automáticamente en el de las variables PERSISTENT.

Después del comando en línea	VAR	VAR RETAIN	VAR PERSISTENT
Restablecer frío	Los valores se reinician	Los valores se mantienen	Los valores se mantienen
Restablecer origen	Los valores se reinician	Los valores se reinician	Los valores se reinician
Descargar	Los valores se reinician	Los valores se mantienen	Los valores se mantienen
Cambio en línea	Los valores se mantienen	Los valores se mantienen	Los valores se mantienen

- [SUPER](#)
- [THIS](#)
- [Variable types - attribute keywords](#)
- [RETAIN: for remanent variables of type RETAIN](#)
- [PERSISTENT: for remanent variables of type PERSISTENT](#)
- [CONSTANT: for constants](#)

6.0.6 Links:

- [🔗 Local Variables - VAR, infosys.beckhoff.com/](https://infosys.beckhoff.com/)
- [🔗 Instance Variables - VAR_INST, infosys.beckhoff.com/](https://infosys.beckhoff.com/)
- [🔗 www.plccoder.com/instance-variables-with-var_inst](https://www.plccoder.com/instance-variables-with-var_inst)
- [🔗 www.plccoder.com/var_temp-var_stat-and-var_const](https://www.plccoder.com/var_temp-var_stat-and-var_const)
- [🔗 Tipos de variables y variables especiales](#)

7. Tipos de Datos


7.1 Declaracion de una Variable:

La declaración de variables en CODESYS ó TwinCAT incluirá: - Un nombre de variable - Dos puntos - Un tipo de dato - Un valor inicial opcional - Un punto y coma - Un comentario opcional

7.1.1 Tipos de Datos:

7.1.2 Las ventajas de las estructuras de datos.

- La principal aportación de las estructuras de datos y los tipos de datos creados por el usuario es la claridad y el orden del código resultante.
-

- Estructuras de datos: (STRUCT)
 -  [Extender una Estructura, Infosys Beckhoff](#)
 - Datos de usuario:UDT (User Data Type) Los UDT (User Data Type) son tipos de datos que el usuario crea a su medida, según las necesidades de cada proyecto.
-

When programming in TwinCAT, you can use different data types or instances of function blocks. You assign a data type to each identifier. The data type determines how much memory space is allocated and how these values are interpreted.

The following groups of data types are available:

7.1.3 Standard data types

TwinCAT supports all data types described in the IEC 61131-3 standard.

- BOOL
- Integer Data Types
- REAL / LREAL
- STRING
- WSTRING
- Time, date and time data types
- LTIME

7.1.4 Extensions of the IEC 61131-3 standard

- BIT
- ANY and ANY_
- Special data types XINT, UXINT, XWORD and PVOID
- REFERENCE
- UNION
- POINTER
- Data type __SYSTEM.ExceptionCode

7.1.5 User-defined data types

Note the recommendations for naming objects.

- POINTER
- REFERENCE
- ARRAY
- Subrange Types User-defined data types that you create as DUT object in the TwinCAT PLC project tree:
- Structure
- Enumerations
- Alias
- UNION

7.1.6 Further Information

- BOOL
- Integer Data Types
- Subrange Types
- BIT
- REAL / LREAL
- STRING
- WSTRING
- Time, date and time data types
- ANY and ANY_
- Special data types XINT, UXINT, XWORD and PVOID
- POINTER
- Data type __SYSTEM.ExceptionCode
- Interface pointer / INTERFACE
- REFERENCE https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2529458827.html&id=
- ARRAY
- Structure
- Enumerations
- Alias
- UNION

7.1.7 Links Tipos de Datos:

- [12. TwinCAT 3: Standard data types](#)
- help.codesys.com, Tipos de datos
- www.infopl.net, codesys-variables
- [TC10.Beckhoff TwinCAT3 DUT](#) .JP

8. Interfaz fluida

8.0.1 Links:

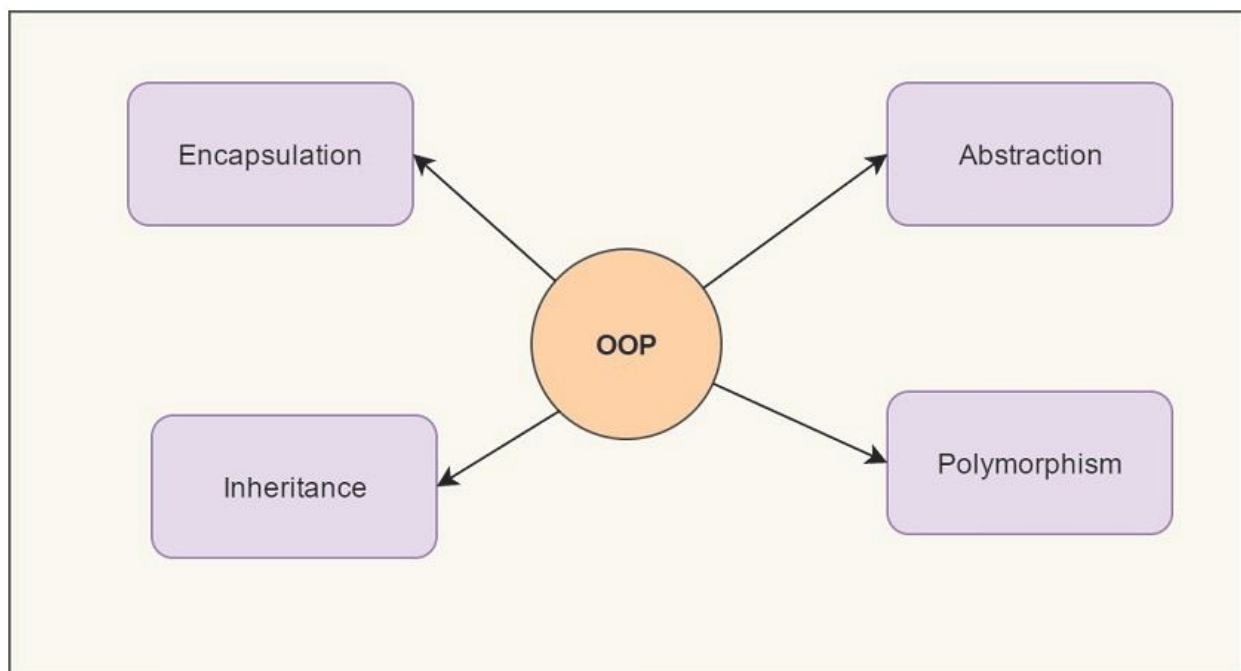
- [fluent-code, www.plccoder.com](#)
- [fluent-interface-and-method-chaining-in-twincat-3](#)

9. Principios OOP

9.1 4 Pilares

Principios OOP: (4 pilares)

- **Abstracción** -- La forma de **plasmear algo hacia el código** para enfocarse en su uso. No enfocarnos tanto en que hay por detras del codigo si no en el uso de este.
- **Encapsulamiento** -- No toda la información de nuestro objeto es **relevante y/o accesible** para el usuario.
- **Herencia** -- Es la cualidad de **heredar características** de otra clase. (EXTENDS)
- **Polimorfismo** -- Las **múltiples formas** que puede obtener un objeto si comparte la misma **clase o interfaz**. (IMPLEMENTS)



Four Pillars of Object Oriented Programming

9.2 Abstracción

abstraccion

Abstraction is the process of hiding important information, showing only the most essential information. It reduces code complexity and isolates the impact of changes. Abstraction can be understood from a real-life example: turning on a television must only require clicking on a button, as people don't need to know or the process that it goes through. Even though that process can be complex and important, there is no need for the user to know how it is implemented. The important information that isn't required is hidden from the user, reducing code complexity, enhancing data hiding and reusability, thus making function blocks easier to implement and modify.

La palabra clave ABSTRACT está disponible para bloques de funciones, métodos y propiedades. Permite la implementación de un proyecto PLC con niveles de abstracción.

La abstracción es un concepto clave de la programación orientada a objetos. Los diferentes niveles de abstracción contienen aspectos de implementación generales o específicos.

Aplicación de la abstracción:

Es útil implementar funciones básicas o puntos en común de diferentes clases en una clase básica abstracta. Se implementan aspectos específicos en subclases no abstractas. El principio es similar al uso de una interfaz. Las interfaces corresponden a clases puramente abstractas que contienen sólo métodos y propiedades abstractas. Una clase abstracta también puede contener métodos y propiedades no abstractos.

Reglas para el uso de la palabra clave ABSTRACT - No se pueden instanciar bloques de funciones abstractas. - Los bloques de funciones abstractas pueden contener métodos y propiedades abstractos y no abstractos. - Los métodos abstractos o las propiedades no contienen ninguna implementación (sólo la declaración). - Si un bloque de función contiene un método o propiedad abstracta, debe ser abstracto. - Los bloques de funciones abstractas deben extenderse para poder implementar los métodos o propiedades abstractos. - Por lo tanto: un FB derivado debe implementar los métodos/propiedades de su FB básico o también debe definirse como abstracto.

-
- [ABSTRACT, www.plccoder.com](#)
 - [ABSTRACION Concepto, Infosys Beckhoff](#)

9.3 Encapsulamiento

encapsulamiento

La encapsulación se utiliza para agrupar datos con los métodos que operan en ellos y para ocultar datos en su interior. una clase, evitando que personas no autorizadas accedan directamente a ella. Reduce la complejidad del código y aumenta la reutilización. La separación del código permite la creación de rutinas que pueden ser reutilizadas en lugar de copiar y pegar código, reduciendo la complejidad del programa principal.

- <https://www.plccoder.com/encapsulation/>

9.4 Herencia

herencia

Inheritance allows the user to create classes based on other classes. The inherited classes can use the base class's functionalities as well as some additional functionalities that the user may define. It eliminates redundant code, prevents copying and pasting and makes expansion easier. This is very useful because it allows classes to be extended or modified (overridden) without changing the base class's code implementation. What do an old landline phone and a smartphone have in common? Both of them can be classified as phones. Should they be classified as objects? No, as they also define the properties and behaviors of a group of objects. A smartphone works just like a regular phone, but it is also able to take pictures, navigate the internet, and do many other things. So, old landline phone and smartphone are child classes that extend the parent phone class.

Superclase: la clase cuyas características se heredan se conoce como superclase (o una clase base o una clase principal).

Subclase: la clase que hereda la otra clase se conoce como subclase (o una clase derivada, clase extendida o clase hija).

- [\[stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance\]](https://stefanhenneken.net/2017/04/23/iec-61131-3-methods-properties-and-inheritance/)(https://stefanhenneken.net/2017/04/23/

9.5 Polimorfismo

polimorfismo

The concept of polymorphism is derived by the combination of two words: Poly (Many) and Morphism (Form). It refactors ugly and complex switch cases/case statements. Object-Oriented PLC Programming 8 Polymorphism allows an object to change its appearance and performance depending on the practical situation in order to be able to carry out a particular task [10]. It can be either static or dynamic: static polymorphism occurs when the object's type is defined by the compiler; dynamic polymorphism occurs when the type is determined during run-time, making it possible for a same variable to access different objects while the program is running. A good example to explain polymorphism is a Swiss Army Knife (Figure 2.4): Figure 2.4 - Swiss Army Knife A Swiss Army Knife is a single tool that includes a bunch of resources that can be used to solve different issues. Selecting the proper tool, a Swiss Army Knife can be used to efficiently perform a certain set of valuable tasks. In the dual way, a simple adder block that adapts itself to cope with, for instance, int, float, string, and time data types is an example of a polymorphic programming resource.

¿Como conseguir el Polimorfismo?

El polimorfismo se puede obtener gracias a las Interfaces y/o las Clases Abstractas.

.Interface: (INTERFACE) - Son un **contrato que obliga** a una clase a **implementar** las **propiedades** y/o **métodos** definidos. - Son una plantilla (sin lógica).

.Clases Abstractas: (ABSTRACT) - Son Clases que no se pueden instanciar, solo pueden ser implementadas a través de la herencia.

• Diferencias:

Clases abstractas	Interfaces
1.- Limitadas a una sola implementación.	1. No tiene limitación de implementación.
2.- Pueden definir comportamiento base.	2. Expone propiedades y métodos abstractos (sin lógica).

### Links Polimorfismo:	

- [polymorphism](#), www.plccoder.com
- [abstract](#), www.plccoder.com
- [\[stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance\]](https://stefanhenneken.net/2017/04/23/stefanhenneken.net/iec-61131-3-methods-properties-and-inheritance)(https://stefanhenneken.net/2017/04/23/
- [AT&U, CODESYS - Runtime polymorphism using inheritance \(OOP\)](#)
- [AT&U,CODESYS - Runtime polymorphism using an ITF \(OOP\)](#)

10. Relaciones

.Relaciones:

Vamos a ver 2 tipos de relaciones:

- Asociación.
- **De uno a uno:** Una clase mantiene una **asociación de a uno** con otra clase.
- **De uno a muchos:** Una clase mantiene una asociación con otra clase **a través de una colección**.
- **De muchos a muchos:** La **asociación se da en ambos lados** a través de una colección.
- Colaboración.
- La colaboración se da **a través de una referencia de una clase** con el fin de **lograr un cometido**.

11. SOLID

11.1 SOLID



Principles of Object Oriented Design

- Propuesta por **Robert C.Martin** en el 2000. - Son **recomendaciones** para escribir un código **sostenible, mantenible, escalable y robusto**. - Beneficios:
- Alta **Cohesión**. Colaboracion entre clases. - Bajo **Acoplamiento**. Evitar que una clase dependa fuertemente de otra clase.
- **Principio de Responsabilidad Única**: Una clase debe tener **una razón** para existir mas no para cambiar.
- **Principio de Abierto/Cerrado**: Las piezas del software deben estar **abiertas para la extensión** pero **cerradas para la modificación**.
- **Principio de Sustitución de Liskov**: Las **clases subtipos** deberían ser reemplazables por sus **clases padres**.
- **Principio de Segregación de Interfaz**: Varias **interfaces** funcionan **mejor que una sola**.
- **Principio de Inversión de Dependencia**: Clases de **alto nivel** no deben depender de las clases **bajo nivel**.

Además de los principios SOLID, existen otros principios como: - Keep It Simple, Stupid (KISS). - Don't Repeat Yourself (DRY). - Law Of Demeter (LOD). - You Ain't Gonna Need It (YAGNI).

Todos estos principios tienen el objetivo común de mejorar la mantenibilidad y la reutilización del software.

Links:

- [🔗 Cómo explicar conceptos de programación orientada a objetos a un niño de 6 años](#)

11.2 Principio de Responsabilidad Única

Principio de Responsabilidad Única:

Links:

- [stefanhenneken.net,iec-61131-3-solid-the-single-responsibility-principle](https://stefanhenneken.net/iec-61131-3-solid-the-single-responsibility-principle)

11.3 Principio de Abierto/Cerrado

Principio de Abierto/Cerrado

La definición del principio abierto/cerrado El Principio Abierto/Cerrado (OCP) fue formulado por Bertrand Meyer en 1988 y establece:

Una entidad de software debe estar abierta a extensiones, pero al mismo tiempo cerrada a modificaciones. Entidad de software: Esto significa una clase, bloque de función, módulo, método, servicio, ...

Abierto: el comportamiento de los módulos de software debe ser extensible.

Cerrado: la capacidad de expansión no debe lograrse cambiando el software existente.

Cuando Bertrand Meyer definió el Principio Abierto/Cerrado (OCP) a fines de la década de 1980, la atención se centró en el lenguaje de programación C++. Usaba herencia, bien conocida en el mundo orientado a objetos. La disciplina de la orientación a objetos, que aún era joven en ese momento, prometía grandes mejoras en la reutilización y la mantenibilidad al permitir que clases concretas se usaran como clases base para nuevas clases.

Cuando Robert C. Martin se hizo cargo del principio de Bertrand Meyer en la década de 1990, lo implementó técnicamente de manera diferente. C++ permite el uso de herencia múltiple, mientras que la herencia múltiple rara vez se encuentra en los lenguajes de programación más nuevos. Por este motivo, Robert C. Martin se centró en el uso de interfaces. Se puede encontrar más información al respecto en el libro (enlace publicitario de Amazon *) [Arquitectura limpia: el manual práctico para el diseño de software profesional](#).

Resumen Sin embargo, adherirse al principio abierto/cerrado (OCP) conlleva el riesgo de un exceso de ingeniería. La opción de extensiones solo debe implementarse donde sea específicamente necesario. El software no puede diseñarse de tal manera que todas las extensiones imaginables puedan implementarse sin realizar ajustes en el código fuente.

!!! este link es en aleman, poner el link en ingles cuando salga:!!!

Links:

- [stefanhenneken.net, iec-61131-3-solid-das-open-closed-principle](#)

11.4 Principio de Sustitución de Liskov

Principio de sustitución de Liskov

Links:

- [stefanhenneken.net,iec-61131-3-solid-the-liskov-substitution-principle](https://stefanhenneken.net/iec-61131-3-solid-the-liskov-substitution-principle)

11.5 Principio de Segregación de Interfaz

Principio de Segregación de Interfaz

Links:

- [stefanhenneken.net,iec-61131-3-solid-the-interface-segregation-principle](https://stefanhenneken.net/iec-61131-3-solid-the-interface-segregation-principle)

11.6 Principio de Inversión de Dependencia

Principio de Inversión de Dependencia:

Links:

- [stefanhenneken.net,iec-61131-3-solid-the-dependency-inversion-principle](https://stefanhenneken.net/iec-61131-3-solid-the-dependency-inversion-principle)

12. UML

12.1 UML

UML

- <https://www.plccoder.com/twincat-uml-class-diagram/>

12.2 Class UML

Class UML

La jerarquía de herencia se puede representar en forma de diagrama. El lenguaje de modelado unificado (UML) es el estándar establecido en esta área. UML define varios tipos de diagramas que describen tanto la estructura como el comportamiento del software.

Una buena herramienta para describir la jerarquía de herencia de bloques de funciones es el diagrama de clases.

Los diagramas UML se pueden crear directamente en TwinCAT 3. Los cambios en el diagrama UML tienen un efecto directo en las POU. Por lo tanto, los bloques de funciones se pueden modificar y modificar a través del diagrama UML.

Cada caja representa un bloque de función y siempre se divide en tres secciones horizontales. La sección superior muestra el nombre del bloque de funciones, la sección central enumera sus propiedades y la sección inferior enumera todos sus métodos. En este ejemplo, las flechas muestran la dirección de la herencia y siempre apuntan hacia el bloque de funciones principal.

Links UML listado de referencias:

- stefanhenneken.net, UML Class
- www.lucidchart.com/tutorial-de-diagrama-de-clases-uml
- www.edrawsoft.com/uml-class-diagram-explained
- blog.visual-paradigm.com/what-are-the-six-types-of-relationships-in-uml-class-diagrams
- [Ingeniería del Software: Fundamentos de UML usando Papyrus](#)
- plantuml.com/class-diagram
- www.planttext.com
- [UML Infosys Beckhoff](#)

12.3 StateChart UML

12.3.1 state chart:

13. Tipos de Diseño para programacion de PLC

13.0.1 Tipos de Diseño para programacion de PLC:

Ingenieria de desarrollo para la programación OOP - Diseño por Componente, Unidad, Dispositivo, Objeto... - Los objetos son las unidades básicas de la programación orientada a objetos. - Un componente proporciona servicios, mientras que un objeto proporciona operaciones y métodos. Un componente puede ser entendido por todos, mientras que un objeto solo puede ser entendido por los desarrolladores. - Las unidades son los grupos de código más pequeños que se pueden mantener y ejecutar de forma independiente - Diseño por Pruebas Unitarias. - Diseño en UML.

Units: (Ejemplo de Unidades): -FCAnalogSensor -FBGenericUnit

!!! puntos que se pueden incluir en el curso!!!

- Basic of Structured Text programming Language
- Modular Design
- Classes
- Methods
- Properties
- Inheritance
- Polymorphism
- Access Specifiers
- Pointers and References
- Interfaces and Abstractions
- Advanced State Pattern
- Wrappers and Features
- Layered Design
- Final Project covering a real-world problem to be solved using OOP
- [Texto estructurado \(ST\)](#), [Texto estructurado extendido \(ExST\)](#)

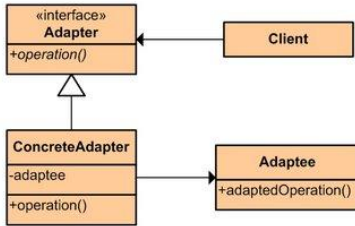
14. Patrones de Diseño

14.1 Patrones de Diseño

PATRONES DE DISEÑO:

Los patrones de diseño son soluciones generales y reutilizables para problemas comunes que se encuentran en la programación de software. En la programación orientada a objetos, existen muchos patrones de diseño que se pueden aplicar para mejorar la modularidad, la flexibilidad y el mantenimiento del código. Algunos ejemplos de patrones de diseño que se pueden aplicar en la programación de PLCs incluyen el patrón Singleton, el patrón Factory Method, el patrón Observer y el patrón Strategy. Por ejemplo, el patrón Singleton se utiliza para garantizar que solo exista una instancia de una clase determinada en todo el programa. Esto puede ser útil en la programación de PLCs cuando se quiere asegurar que solo hay una instancia activa del objeto que controla un determinado proceso o dispositivo. El patrón Factory Method se utiliza para crear instancias de objetos sin especificar explícitamente la clase concreta a instanciar. Esto puede ser útil en la programación de PLCs cuando se quiere crear objetos según las necesidades específicas del programa. El patrón Observer se utiliza para establecer una relación uno a muchos entre objetos, de manera que cuando un objeto cambia su estado, todos los objetos relacionados son notificados automáticamente. Este patrón puede ser muy útil en la programación de PLCs para establecer relaciones entre diferentes componentes del sistema, como sensores y actuadores. El patrón Strategy se utiliza para definir un conjunto de algoritmos intercambiables, y luego encapsular cada uno como un objeto. Este patrón puede ser útil en la programación de PLCs cuando se desea cambiar dinámicamente el comportamiento del sistema según las condiciones del entorno. En resumen, los patrones de diseño son una herramienta muy útil para mejorar la calidad del código en la programación de PLCs y se pueden aplicar con éxito en la programación orientada a objetos para PLCs.

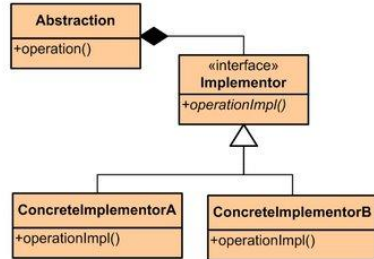
```
1  "Los patrones de diseño son
2  descripciones de objetos y clases
3  conectadas que se personalizan para
4  resolver un problema de diseño
5  general en un contexto particular".
6  - Gang of Four
```



Adapter

Type: Structural

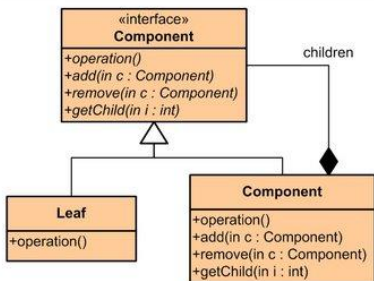
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.



Bridge

Type: Structural

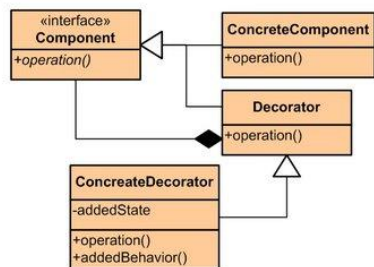
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.



Composite

Type: Structural

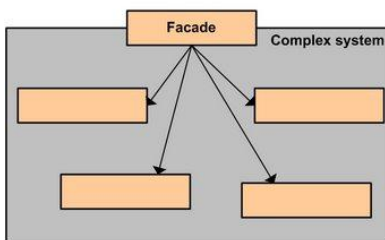
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Decorator

Type: Structural

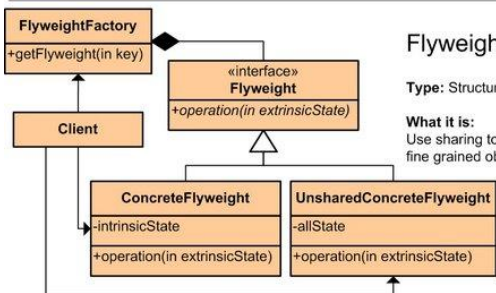
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.



Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



Flyweight

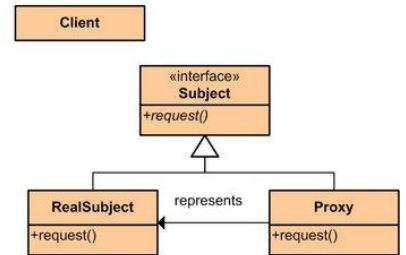
Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Proxy

Type: Structural

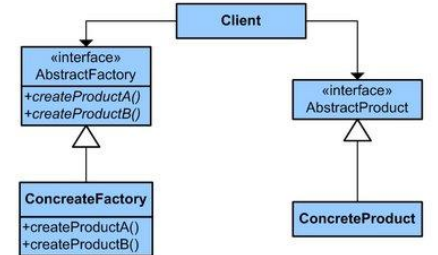
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Abstract Factory

Type: Creational

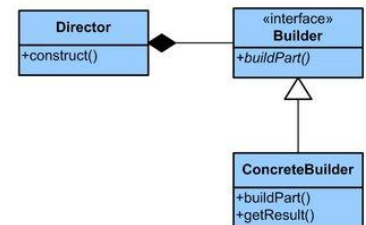
What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Builder

Type: Creational

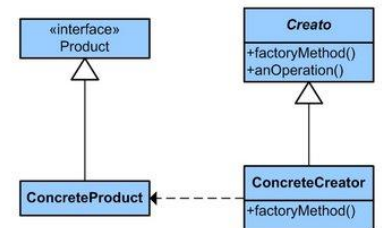
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Factory Method

Type: Creational

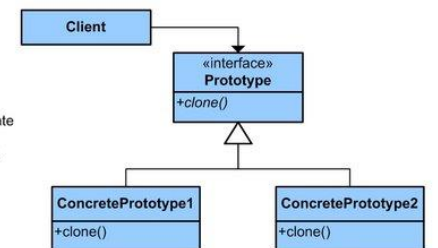
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Singleton

Type: Creational

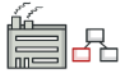
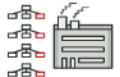

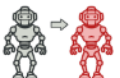

What it is:
Ensure a class only has one instance and provide a global point of access to it.



The Catalog of Design Patterns

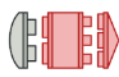


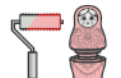

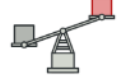

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

	
Factory Method	Abstract Factory
	
Builder	Prototype
	
Singleton	




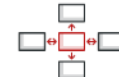
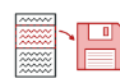

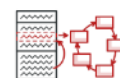
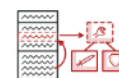


Structural patterns

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

	
Adapter	Bridge
	
Composite	Decorator
	
Facade	Flyweight
	
Proxy	

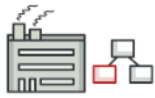
Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

			
Chain of Responsibility	Command	Iterator	Mediator
			
Memento	Observer	State	Strategy
			
Template Method	Visitor		

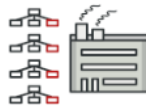
Patrones de diseño creacional

Los patrones de diseño creacional proporcionan varios mecanismos de creación de objetos, que aumentan la flexibilidad y la reutilización del código existente.



Método de fábrica

Proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases modifiquen el tipo de objetos que se crearán.



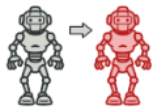
Fábrica abstracta

Le permite producir familias de objetos relacionados sin especificar sus clases concretas.



Constructor

Le permite construir objetos complejos paso a paso. El patrón le permite producir diferentes tipos y representaciones de un objeto utilizando el mismo código de construcción.



Prototipo

Le permite copiar objetos existentes sin hacer que su código dependa de sus clases.

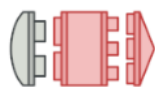


único

Le permite asegurarse de que una clase tenga solo una instancia, mientras proporciona un punto de acceso global a esta instancia.

Patrones de diseño estructural

Los patrones de diseño estructural explican cómo ensamblar objetos y clases en estructuras más grandes, manteniendo estas estructuras flexibles y eficientes.



Adaptador

Permite que objetos con interfaces incompatibles colaboren.



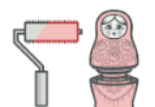
Puente

Le permite dividir una clase grande o un conjunto de clases estrechamente relacionadas en dos jerarquías separadas, abstracción e implementación, que se pueden desarrollar de forma independiente.



Compuesto

Le permite componer objetos en estructuras de árbol y luego trabajar con estas estructuras como si fueran objetos individuales.



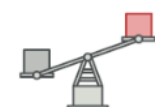
Decorador

Le permite adjuntar nuevos comportamientos a los objetos colocando estos objetos dentro de objetos contenedores especiales que contienen los comportamientos.



Fachada

Proporciona una interfaz simplificada para una biblioteca, un marco o cualquier otro conjunto complejo de clases.



peso mosca

Le permite colocar más objetos en la cantidad disponible de RAM al compartir partes comunes del estado entre múltiples objetos en lugar de mantener todos los datos en cada objeto.



Apoderado

Le permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, lo que le permite realizar algo antes o después de que la solicitud llegue al objeto original.

Patrones de diseño de comportamiento

Los patrones de diseño de comportamiento se ocupan de los algoritmos y la asignación de responsabilidades entre objetos.



Cadena de responsabilidad

Le permite pasar solicitudes a lo largo de una cadena de controladores. Al recibir una solicitud, cada controlador decide procesarla o pasarla al siguiente controlador de la cadena.



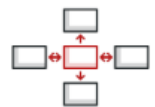
Dominio

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación le permite pasar solicitudes como argumentos de método, retrasar o poner en cola la ejecución de una solicitud y admitir operaciones que se pueden deshacer.



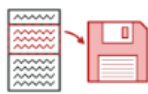
iterador

Le permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).



Mediator

Le permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos y los obliga a colaborar solo a través de un objeto mediador.



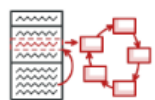
Recuerdo

Le permite guardar y restaurar el estado anterior de un objeto sin revelar los detalles de su implementación.



Observador

Le permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



Expresar

Permite que un objeto altere su comportamiento cuando cambia su estado interno. Parece como si el objeto cambiara su clase.



Estrategia

Le permite definir una familia de algoritmos, poner cada uno de ellos en una clase separada y hacer que sus objetos sean intercambiables.



Método de plantilla

Define el esqueleto de un algoritmo en la superclase pero permite que las subclases anulen pasos específicos del algoritmo sin cambiar su estructura.



Visitante

Le permite separar los algoritmos de los objetos en los que operan.

Clasificación según su propósito: Los patrones de diseño se clasificaron originalmente en tres grupos:

- Creacionales.
- Estructurales.
- De comportamiento.

Clasificación según su ámbito:

- De clase: Basados en la herencia de clases.
- De objeto: Basados en la utilización dinámica de objetos.

Patrones creacionales:

- Builder
- Singleton
- Dependency Injection
- Service Locator
- Abstract Factory
- Factory Method

Patrones estructurales:

- Adapter
- Data Access Object (DAO)
- Query Object
- Decorator
- Bridge

Patrones de comportamiento:

- Command
- Chain of Responsibility
- Strategy
- Template Method
- Interpreter
- Observer
- State
- Visitor
- Iterator

Links de Patrones de Diseño:

- [IEC 61131-3: SOLID – The Interface Segregation Principle](#)

14.2 Patron de Estrategia

TwinCAT with Head First Design Patterns Ch.1 - IntroStrategy Pattern.docx

15. Links

15.0.1 Links

15.0.2 Mención a la Fuentes Links empleadas para la realización de esta Documentación:
