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.

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...

Link al Video de Youtube 001:

- 🔗 001 - OOP IEC 61131-3 PLC -- Introducción a la pagina de documentación SSG, repositorio...

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Introduccion

📖 Curso Programación Orientada a Objetos Youtube -- OOP :

by Runtimevic -- Víctor Durán Muñoz.

¿ 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.

¿Cómo pensar en Objetos?

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

```
📱 Ejemplo: (Telefono móvil-smartphone)

. ¿Qué atributos (Propiedades) reconocemos?
   - color.
   - marca.
. ¿Qué se puede hacer? (Metodos)
   - Realizar llamadas.
   - Navegar por internet.
```

```
🚗 Ejemplo: (Coche)

. ¿Qué atributos (Propiedades) reconocemos?
   - color.
   - marca.
. ¿Qué se puede hacer? (Metodos)
   - conducir.
   - frenar.
   - acelerar.
```

Links:

- 🔗 Codesys admite OOP
- 🔗 Beckhoff TwinCAT 3 admite OOP
- 🔗 Why Object Oriented PLC Programming is Essential for Industrial Automation
- 🔗 cascadingsoft.com,object-oriented-programming-oop-concepts-benefit
- 🔗 object-oriented-programming-in-plc-ooip-or-oops-supported-plc
- 🔗 The Benefits of OOP in PLC Programming: A Case Study with a Conveyor System
- 🔗 Fundamental Concepts of Object Oriented Programming
- 🔗 www.genbeta.com, programacion-orientada-a-objetos-principales-conceptos-explicados-para-todos-publicos

Link al Video de Youtube 001:

- 🔗 001 - OOP IEC 61131-3 PLC -- Introducción a la pagina de documentación SSG, repositorio...

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Tipos de paradigmas

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 inico 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**.

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

Ventajas de la Programación OOP:

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

Link Tipos de Paradigmas:

- 🔗 Programación funcional ¿Qué es y cómo sacarle partido?

Link al Video de Youtube 002 y 003:

- 🔗 002 - OOP IEC 61131-3 PLC -- Clase y Objeto
- 🔗 003 - OOP IEC 61131-3 PLC -- Clase y Objeto

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Tipos de Datos

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

```
( <pragma> )*
<scope> ( <type qualifier> )?
    <identifier> (AT <address> )? : <data type> ( := <initial value> )? ;
END_VAR
```

- 🔗 infosys.beckhoff.com, Declaring variables

```
VAR
   nVar1   : INT := 12;                  // initialization value 12
   nVar2   : INT := 13 + 8;              // initialization value defined by an expression of constants
   nVar3   : INT := nVar2 + F_Fun(4);      //initialization value defined by an expression that contains a function call; notice the order!
   pSample : POINTER TO INT := ADR(nVar1); //not described in the standard IEC61131-3: initialization value defined by an adress function; Notice: the pointer will not be initialized during an Online Change
END_VAR
```

🔗 Tipos de Datos:

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)

- 🔗 19. TwinCAT 3: Structures
- 🔗 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:

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

Extensions of the IEC 61131-3 standard

- BIT
- ANY and ANY_
- http://soup01.com/ja/2023/06/13/post-8579/
- Special data types XINT, UXINT, XWORD and PVOID
- REFERENCE
- UNION
- POINTER

- Data type __SYSTEM.ExceptionCode
- T_ARG
- https://infosys.beckhoff.com/content/1033/tcplclib_tc2_utilities/35376139.html?id=4303460770811510149
- http://soup01.com/ja/2023/06/23/post-8605/

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

Further Information

- BOOL
- Integer Data Types
- Subrange Types
- BIT
- REAL / LREAL
- STRING
- WSTRING
- Time, date and time data types
- ANY and ANY_
- http://soup01.com/ja/2023/06/13/post-8579/
- 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
- 🔗 Special data types XINT, UXINT, XWORD and PVOID

Links Tipos de Datos:

- 🔗 12. TwinCAT 3: Standard data types
- 🔗 help.codesys.com, Tipos de datos
- 🔗 www.infoplc.net, codesys-variables
- 🔗 TC10.Beckhoff TwinCAT3 DUT .JP

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Tipos de variables y variables especiales

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.

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.
  - En TwinCAT build 4026 existe la sobrecarga de las VAR_INPUT, en su declaración se podrán incializar las variables declaradas de esta forma al

llamar al FB, FC, metodo, etc..., no es obligatorio incluirla en la llamada ya que tendrán el valor por defecto si no se llama en su modulo correspondiente.

- 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 tambien tiene que estar declarada la misma variable con el mismo nombre en una GVL (Lista de Varaibles 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.

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'

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 RETEAIN 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.

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 reinicializan	Los valores se mantienen	Los valores se mantienen
Restablecer origen	Los valores se reinicializan	Los valores se reinicializan	Los valores se reinicializan
Descargar	Los valores se reinicializan	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
  - VAR_GENERIC CONSTANT

- En la versión de TwinCAT build 4026: podremos declarar VAR_GENERIC CONSTANT ver imagen adjunta:

Links:

- 🔗 Local Variables - VAR, infosys.beckhoff.com/
- 🔗 Instance Variables - VAR_INST, infosys.beckhoff.com/
- 🔗 www.plccoder.com/instance-variables-with-var_inst
- 🔗 www.plccoder.com/var_temp-var_stat-and-var_const
- 🔗 Tipos de variables y variables especiales

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Modificadores de acceso

Modificadores de Acceso:

- PUBLIC:
  - Son accesibles luego de instanciar la clase.
  - Corresponde a la especificación de modificador sin restricción de acceso.
- PRIVATE:
  - Son accesibles solo dentro de la clase.
  - El acceso está restringido al bloque de funciones Heredado y en el programa MAIN, respectivamente.
- PROTECTED:
  - Son accesibles dentro de la clase.
  - Son accesibles a través de la herencia.
  - El acceso está restringido, no se puede acceder desde el programa principal, desde el MAIN.
- INTERNAL:
  - El acceso está limitado al espacio de nombres (la biblioteca).
- FINAL:
  - No se permite sobrescribir, en un derivado del bloque de funciones.
  - Esto significa que no se puede sobrescribir/extender en una subclase posiblemente existente.

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Clases y Objetos

Clases y Objetos:

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

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

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

```
FUNCTION_BLOCK Coche
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    _Marca : STRING;
    _Color : STRING;
    accion : STRING;
END_VAR
-----------------------------------------------------------------
METHOD PUBLIC Acelerar
accion := 'acelerar';
-----------------------------------------------------------------
METHOD PUBLIC Conducir
accion := 'conducir';
-----------------------------------------------------------------
METHOD PUBLIC Frenar
accion := 'frenar';
-----------------------------------------------------------------
PROPERTY PUBLIC Color : STRING
Get
    Color := _Color;
Set
    _Color := Color;
-----------------------------------------------------------------
PROPERTY PUBLIC Marca : STRING
Get
    Marca := _Marca;
Set
    _Marca := Marca;
```

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

```
PROGRAM _01_Clase_y_Objetos
VAR
   // tenemos la Clase Coche y la instanciamos y obtenemos los Objetos: Mercedes,
Bmw y Audi.
   Mercedes : Coche;
   Bmw : Coche;
   Audi: Coche;

   Color : STRING;
   Marca : STRING;

   Acelerar : BOOL;
   Conducir:  BOOL;
   Frenar  : BOOL;
END_VAR

//Objeto Mercedes
//llamadas a sus métodos.
IF Acelerar THEN
   Mercedes.Acelerar();
   Acelerar := FALSE;
END_IF

IF Conducir THEN
   Mercedes.Conducir();
   Conducir := FALSE;
END_IF

IF Frenar THEN
   Mercedes.Frenar();
   Frenar := FALSE;
END_IF

//llamadas a sus propiedades.
Mercedes.Marca := 'Mercedes';
Mercedes.Color := 'Negro';
Color := Mercedes.Color;
```

Links:

- 🔗 methods-properties-and-inheritance (stefanhenneken)
- 🔗 Object Oriented Industrial Programming (OOIP) -- March 2021 CODESYS Tech Talk
- 🔗 Clase de Programación Orientada a Objetos en una hora ! Cómo crear tus entidades, herencia y más
- 🔗 Object Oriented Programming (OOP) in Java Course

Link al Video de Youtube 002:

- 🔗 002 - OOP IEC 61131-3 PLC -- Clase y Objeto

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Bloques de Funciones

Declaracion de un Function Block:

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

Implementación Bloque de Funciones:

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:

```
FUNCTION_BLOCK INTERNAL ABSTRACT FB
FUNCTION_BLOCK INTERNAL FINAL FB
FUNCTION_BLOCK PUBLIC FINAL FB
FUNCTION_BLOCK ABSTRACT FB
FUNCTION_BLOCK PUBLIC ABSTRACT FB
FUNCTION_BLOCK FB EXTENDS FB1 IMPLEMENTS Interface1, Interface2, Interface3
```

Link al Video de Youtube 003:

- 🔗 003 - OOP IEC 61131-3 PLC -- Clase y Objeto

Link al Video de Youtube 004:

- 🔗 004 - OOP IEC 61131-3 PLC -- Clase y Objeto

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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

```
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).

Link al Video de Youtube 004:

- 🔗 004 - OOP IEC 61131-3 PLC -- Clase y Objeto

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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**:

```
FUNCTION_BLOCK fb_tipos_de_datos
VAR_INPUT
   binput : BOOL;
END_VAR
VAR_INPUT
   binput2 : BOOL;
END_VAR
VAR_OUTPUT
   output1 : REAL;
END_VAR
VAR_IN_OUT
   in_out1 : LINT;
END_VAR
VAR_IN_OUT CONSTANT
   in_out_constant1 : DINT;
END_VAR
VAR
   var1 : STRING;
END_VAR
VAR_TEMP
   temp1 : ULINT;
END_VAR
VAR_STAT
   nVarStat1 : INT;
END_VAR
VAR_EXTERNAL
   nVarExt1 : INT; // 1st external variable
END_VAR
VAR PERSISTENT
   nVarPers1 : DINT; (* 1. Persistent variable *)
   bVarPers2 : BOOL; (* 2. Persistent variable *)
END_VAR
VAR RETAIN
   nRem1 : INT;
END_VAR
VAR CONSTANT
   n : INT:= 10;
END_VAR
```

Link al Video de Youtube 004:

- 🔗 004 - OOP IEC 61131-3 PLC -- Clase y Objeto

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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

Link al Video de Youtube 003:

- 🔗 003 - OOP IEC 61131-3 PLC -- Clase y Objeto

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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:

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

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

Ejemplo de declaración de METHOD:

```
METHOD Method1 : BOOL
VAR_INPUT
    nIn1  : INT;
    bIn2  : BOOL;
END_VAR
VAR_OUTPUT
    fOut1 : REAL;
    sOut2 : STRING;
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
- 🔗 Tutorial #18: Einstieg in die objektorientierte Programmierung mit CoDeSys - Teil 1: Methoden

Link al Video de Youtube 005:

- 🔗 005 - OOP IEC 61131-3 PLC -- Objeto Metodo

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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:
  - Son accesibles solo dentro de su propia Clase (Bloque de Función).
  - Sin acceso desde la clase heredada.
  - Sin acceso desde el programa principal, desde el MAIN.
- PROTECTED:
  - Accesible desde dentro de su propia Clase.
  - Accesible desde clases heredadas.
  - El acceso está restringido,no se puede acceder desde el programa principal, desde el MAIN.
- 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.

Link al Video de Youtube 005:

- 🔗 005 - OOP IEC 61131-3 PLC -- Objeto Metodo

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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**:

```
METHOD metodo0_Declaracion_variables
VAR_INPUT
   binput : BOOL;
END_VAR
VAR_INPUT
   binput2 : BOOL;
END_VAR
VAR_OUTPUT
   output1 : REAL;
END_VAR
VAR_IN_OUT
   in_out1 : LINT;
END_VAR
VAR_IN_OUT CONSTANT
   in_out_constant1 : DINT;
END_VAR
VAR
   var1 : STRING;
END_VAR
//!!! no se pueden declarar variables TEMPORALES dentro de la zona de
declaración de variables del método!!!
//VAR_TEMP
//  temp1 : ULINT;
//END_VAR
VAR_INST
   counter : INT;
END_VAR
VAR_STAT
   nVarStat1 : INT;
   aarray  : ARRAY[1..n] OF INT;
END_VAR
VAR_EXTERNAL
   nVarExt1 : INT; // 1st external variable
END_VAR
//!!! no se pueden declarar variables PERSISTENT ni RETAIN dentro de la zona
de declaración de variables del método!!!
//VAR PERSISTENT
//    nVarPers1 : DINT; (* 1. Persistent variable *)
//    bVarPers2 : BOOL; (* 2. Persistent variable *)
//END_VAR
//VAR RETAIN
//    nRem1 : INT;
//END_VAR
VAR CONSTANT
   n : INT:= 10;
END_VAR
```

Link al Video de Youtube 005:

- 🔗 005 - OOP IEC 61131-3 PLC -- Objeto Metodo

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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:

```
METHOD Method1 : BOOL
METHOD Method1 : INT
METHOD Method1 : REAL
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:

```
REFERENCE TO <structured type>
//en lugar de simplemente
<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):

```
TYPE ST_Sample :
STRUCT
   bVar  : BOOL;
   nVar  : INT;
END_STRUCT
END_TYPE
```

Declaración del bloque de funciones **FB_Sample**:

```
FUNCTION_BLOCK FB_Sample
VAR
   stLocal     : ST_Sample;
END_VAR
```

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

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

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

```
MyProp REF= stLocal;
```

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

```
stLocal := MyProp;
```

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

```
PROGRAM MAIN
VAR
   fbSample    : FB_Sample;
   nSingleGet  : INT;
   stGet       : ST_Sample;
   bSet        : BOOL;
   stSet       : ST_Sample;
END_VAR
// Get - single member and complete structure possible
nSingleGet := fbSample.MyProp.nVar;
stGet      := fbSample.MyProp;

// Set - only complete structure possible
IF bSet THEN
   fbSample.MyProp REF= stSet;
   bSet            := FALSE;
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.

```
VAR
   fbSample    : FB_Sample;
   nSingleGet  : INT;
END_VAR
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.

```
VAR
   fbSample    : FB_Sample;
   stGet       : ST_Sample;
   nSingleGet  : INT;
END_VAR
stGet      := fbSample.MyProp;
nSingleGet := stGet.nVar;
```

Retorno por INTERFACE:

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

```
METHOD Method1 : interface1
```

Retorno por FUNCTION_BLOCK:

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

```
METHOD Method1 : FB1
```

Link al Video de Youtube 005:

- 🔗 005 - OOP IEC 61131-3 PLC -- Objeto Metodo

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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:

```
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 dentro de su propia Clase (Bloque de Funciones).
- PROTECTED:
  - El acceso está restringido,no se puede acceder desde el programa principal, desde el MAIN.
- 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
- 🔗 Nguyễn Vỹ, Beckhoff & PLC - OOP Properties| TWINCAT3

Link al Video de Youtube 006:

- 🔗 006 - OOP IEC 61131-3 PLC -- Objeto Propiedad

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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:

```
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:

```
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() .

```
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:

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

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

Clasificación de Tipos de Herencia:

- Existen 2 Tipos de Herencia: (Clasificación segun la forma en como hereda una Clase):
- Herencia Simple:
  - Es cuando una clase Hereda únicamente de una Clase.
  - Se pueden establecer niveles de Herencia.
  - La Herencia Simple Si se admite en el IEC61131-3, en Codesys y en TwinCAT.
- Herencia Multiple:
  - Es cuando una clase puede Heredar de 2 clases ó más al mismo tiempo.
  - No importando los niveles de Herencia.
  - La Herencia Multiple No se admite en el IEC61131-3, no se admite ni en Codesys ni en TwinCAT.

Links Herencia Bloque de Funcion:

- 🔗 stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance
- 🔗 Simple Codesys OOP - Inheritance
- 🔗 TC11.Beckhoff TwinCAT3 Function Block Extend.JP
- 🔗 Tutorial #19: Einstieg in die objektorientierte Programmierung mit CoDeSys - Teil 2: Vererbung

Link al Video de Youtube 007:

- 🔗 007 - OOP IEC 61131-3 PLC -- Herencia FB

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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:

```
TYPE ST_Base1 :
STRUCT
   bBool1: BOOL;
   iINT : INT;
   rReal : REAL;
END_STRUCT
END_TYPE
```

```
TYPE ST_Sub1 EXTENDS ST_Base1:
STRUCT
   ttime :TIME;
   tton  : TON;
END_STRUCT
END_TYPE
```

```
TYPE ST_Sub2 EXTENDS ST_Sub1 :
STRUCT
   bBool2: BOOL; // No se podria llamar la variable bBool1 porque la tenemos
declarada en la estructura ST_Base1
END_STRUCT
END_TYPE
```

```
PROGRAM MAIN
VAR
   stestructura1  : ST_Sub1;
   stestructura2  : ST_Sub2;
END_VAR

//Extensión de Estructura:
stestructura1.bBool1;
stestructura1.iINT;
stestructura1.rReal;
stestructura1.ttime;
stestructura1.tton(in:= TRUE, pt:=T#1S);

stestructura2.bBool1;
stestructura2.iINT;
stestructura2.rReal;
stestructura2.ttime;
stestructura2.tton(in:= TRUE, pt:=T#1S);
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.
- Tambien sin usar EXTENDS para la Estructura podriamos realizarlo de la siguiente forma:

```
TYPE ST_2 :
STRUCT
   bBool : BOOL;
END_STRUCT
END_TYPE
```

```
TYPE ST_1:
STRUCT
   sStruct : ST_2;
   sstring : STRING(80);
END_STRUCT
END_TYPE
```

```
PROGRAM MAIN
VAR
   stestructura11 : ST_1;
END_VAR

stestructura11.sstring;
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:

```
TYPE ST_Sub EXTENDS ST_Base1,ST_Base2 :
STRUCT
```

Links:

- 🔗 infosys.beckhoff.com, Extends Structure
- 🔗 help.codesys.com, Structure
- 🔗 help.codesys.com, Structure
- 🔗 help.codesys.com, Structure

Link al Video de Youtube 008:

- 🔗 008 - OOP IEC 61131-3 PLC -- Herencia Estructura e Interface

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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:

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

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

```
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
- 🔗 help.codesys.com, Extends Interface

Link al Video de Youtube 008:

- 🔗 008 - OOP IEC 61131-3 PLC -- Herencia Estructura e Interface

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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.

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

Ejemplos:

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

```
FUNCTION_BLOCK FB_A
VAR_INPUT
   nVarA: INT;
END_VAR

nVarA := 1;

FUNCTION_BLOCK FB_B EXTENDS FB_A
VAR_INPUT
   nVarB : INT := 0;
END_VAR

nVarA := 11;
nVarB := 2;

METHOD DoIt : BOOL
VAR_INPUT
END_VAR
VAR
   nVarB : INT;
END_VAR

nVarB := 22; // Se establece la variable local nVarB.
THIS^.nVarB := 222; // La variable del bloque de funciones nVarB se establece
aunque nVarB está oculta.

PROGRAM MAIN
VAR
  fbMyfbB : FB_B;
END_VAR

fbMyfbB(nVarA:=0, nVarB:= 0);
fbMyfbB.DoIt();
```

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

```
FUNCTION F_FunA : INT
VAR_INPUT
    fbMyFbA : FB_A;
END_VAR
...;

FUNCTION_BLOCK FB_A
VAR_INPUT
    nVarA: INT;
END_VAR
...;

FUNCTION_BLOCK FB_B EXTENDS FB_A
VAR_INPUT
    nVarB: INT := 0;
END_VAR

nVarA := 11;
nVarB := 2;

METHOD DoIt : BOOL
VAR_INPUT
END_VAR
VAR
    nVarB: INT;
END_VAR

nVarB := 22;  //Se establece la variable local nVarB.
F_FunA(fbMyFbA := THIS^);  //F_FunA es llamado via THIS^.

PROGRAM MAIN
VAR
    fbMyFbB: FB_B;
END_VAR

fbMyFbB(nVarA:=0 , nVarB:= 0);
fbMyFbB.DoIt();
```

Links THIS^ pointer:

- 🔗 THIS puntero Infosys Beckhoff
- 🔗 help.codesys.com, THIS
- 🔗 stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance

Link al Video de Youtube 009:

- 🔗 009 - OOP IEC 61131-3 PLC -- Punteros THIS^ y SUPER^

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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.

```
SUPER^();                // Llamada del cuerpo FB de la clase base.
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:

```
FUNCTION_BLOCK FB_Base
VAR_OUTPUT
    nCnt : INT;
END_VAR
```

Metodo -- FB_Base.METH_DoIt:

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

Metodo -- FB_Base.METH_DoAlso:

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

Bloque de Función -- FB_1:

```
FUNCTION_BLOCK FB_1 EXTENDS FB_Base
VAR_OUTPUT
    nBase: INT;
END_VAR
THIS^.METH_DoIt();      // llamada al metodo METH_DoIt del FB_1.
THIS^.METH_DoAlso();

SUPER^.METH_DoIt();     // llamada al metodo METH_DoIt del FB_Base.
SUPER^.METH_DoAlso();
nBase := SUPER^.nCnt;
```

Metodo -- FB_1.METH_DoIt:

```
METHOD METH_DoIt : BOOL
nCnt := 1111;
METH_DoIt := TRUE;
```

Metodo -- FB_1.METH_DoAlso:

```
METHOD METH_DoAlso : BOOL
nCnt := 123;
METH_DoAlso := FALSE;
```

Programa MAIN:

```
PROGRAM MAIN
VAR
    fbMyBase : FB_Base;
    fbMyFB_1 : FB_1;
    nTHIS    : INT;
    nBase    : INT;
END_VAR
fbMyBase();
nBase := fbmyBase.nCnt;
fbMyFB_1();
nTHIS := fbMyFB_1.nCnt;
```

Links SUPER^ pointer:

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

Link al Video de Youtube 009:

- 🔗 009 - OOP IEC 61131-3 PLC -- Punteros THIS^ y SUPER^

December 16, 2023 10:31:59

December 16, 2023 10:31:59

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_3.5.13.0 Objeto Interface
- 🔗 Codesys_3.5.14.0 Objeto Interface
- 🔗 Codesys Implementando Interfaces
- 🔗 Beckhoff Objeto Interface
- 🔗 Beckhoff Implementando Interfaces
- 🔗 Extender Interfaces, Infosys Beckhoff
- 🔗 TC09.Beckhoff TwinCAT3 Function Block-Part4 Interface.JP
- 🔗 Tutorial #20: Einstieg in die objektorientiert Programmierung mit CoDeSys - Teil 3: Interfaces

Link al Video de Youtube 010:

- 🔗 010 - OOP IEC 61131-3 PLC -- Interface

December 16, 2023 10:31:59

December 16, 2023 10:31:59

puntero y referencia

Puntero y Referencia:

En la programación orientada a objetos (OOP) en PLC IEC 61131-3, los punteros y las referencias son dos conceptos importantes que se utilizan para acceder a los datos y métodos de un objeto. Un puntero es una variable que almacena la dirección de memoria de otra variable. Una referencia es una variable que se utiliza para acceder a otra variable sin tener que conocer su dirección de memoria.

. ¿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:

```
// 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
// 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
// variables del tipo correspondiente. también hay que declarar todas las variables -punteros- que contendrán esas direcciones de
// memoria y su correspondiente tipo.
VAR
   stTest1 : stTipo1; //Declara una estructura de datos del tipo stTipo1.

   pin01 : POINTER TO INT; //Declara un puntero para acceder a variables del tipo INT.
   ps20  : POINTER TO STRING[20]; //Declara un puntero para acceder a variables del tipo STRING de 20 caracteres.
   pa20 : POINTER TO ARRAY [1..20] OF INT; //Declara un puntero para acceder a variables del tipo ARRAY de 20 elementos del tipo INT.
   pDword : POINTER TO DWORD; //Declara un puntero para acceder a variables del tipo DWORD.
   past1  : POINTER TO stTipo1; //Declara un puntero para acceder a variables del tipo stTipo1.
   pReal  : POINTER TO REAL; //Declara un puntero para acceder a variables del tipo REAL.
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:

```
PROGRAM SR_Main_02
VAR
   in01    : INT; //Declaración de la variable in01 de tipo entero.
   in02    : INT := 123; //Declaración e inicialización de la variable in02 de tipo
entero.
   in03    : INT; //Declaración de la variable in03 de tipo entero.

   pint    : POINTER TO
INT; //Declaración de un puntero para acceder a variables del tipo entero.
END_VAR

// Ejemplo de uso básico de los operadores ADR y del operador de contenido ^
// Se muestra como asignar a un puntero la dirección de memoria de una variable y
como leer/escribir
// así como un ejemplo de acceso a variables locales de otros programas.

pint := ADR(in01); //Asignamos al puntero la dirección de memoria donde se ubica la
variable in01.
pint^ := 44;     //A la posición de memoria indicada por el puntero, le asignamos el
valor 44
                //Por tanto a la variable in01 se le ha escrito el valor 44.

in02 := in01; // in02 será igual a 44.

pint := ADR(in02); //Cambiamos la dirección para acceder a la dirección de la
variable in02.
in03 := pint^; // in03 tomara el valor del contenido de la posición de memoria
contenida en el
            // que hemos asignado la dirección de in02, por tanto in03= 123.

pint := ADR(SR_Main_01.inLocalAway); //Cargamos la dirección de memoria de una
variable local de
                                // otro programa, la que sería inaccesible por otros medios.
pint^ := 240 ; // La varible 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:

```
PROGRAM SR_Main_01
VAR
   aR20: ARRAY[1..20] OF REAL; //Declara un array de 20 elementos del tipo REAL.
   inIndex: INT; //Declara la variable de indice del array para el acceso indirecto
   xNewVal: BOOL; //Indica que hay una nueva lectura del sensor de fuerza.
   rFuerza: REAL; //Valor de fuerza del sensor.
END_VAR

// Ejemplo01: Se asigna valores del 1 al 20 a cada elemento del array mediante un bucle.
FOR inIndex:=1 TO 20 BY 1 DO  // Se empieza por el valor de la variable indice a 1, hasta 20
   aR20[inIndex] := inIndex; // Al elemento aR20[inIndex] se le asigna el valor de inIndex
END_FOR;                    // Se incrementa inIndex y se repite el proceso.

// Ejemplo02: Creamos un FIFO en el que guardamos un valor analógico de fuera a cada impulso de la señal xNewVal.
IF xNewVal THEN             // Si hay un nuevo valor de fuerza realizamos el codigo.
   xNewVal := FALSE;        // Reset de la señal xNewVal.

   FOR inIndex:=20 TO 2 BY -1 DO // Variable indice a 20, hasta 2.
      aR20[inIndex] := aR20[inIndex-1]; //Desplazamiento de los valores en el FIFO --->
   END_FOR;                      // Se decrementa inIndex y se repite el proceso.

   aR20[1] := rFuerza; // Entrada del valor de fuerza en el primer elemento del FIFO.
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:

```
PROGRAM SR_Main_03
VAR
   stMotor_01 : stMotorCtrl; // Estructura de control del motor 1
   stMotor_02 : stMotorCtrl; // Estructura de control del motor 2
   stMotor_03 : stMotorCtrl; // Estructura de control del motor 3
   pstMotorCtrl : POINTER TO stMotorCtrl; // Puntero para acceder a estructuras del
tipo stMotorCtrl.
   xMarcha : BOOL; // Pulsador marcha motores.
END_VAR

// Ejemplo básico de como acceder a estructuras de datos mediante punteros.
// La estructura de datos empleada es una llamada a stMotorCtrl, que coincide un bit
de marcha, otro de paro y
// valores de velocidad en Rpms y tiempo de aceleración/deceleración.

// Asignamos valores a la estructura para el control del motor 1.

stMotor_01.rTpoAcelDecel := 5.4; // Tiempo para acelerar/decelerar hasta alcanazar
la velocidad.
stMotor_01.rVelRpm := 1436.2; // Velocidad en RPM.
stMotor_01.xMotorOff := TRUE; // Bit de paro ON.
stMotor_01.xMotorOn := FALSE   ; // Bit de marcha OFF.

pstMotorCtrl := ADR(stMotor_01); // Cargamos la dirección de memoria de la
estructura del motor 1
stMotor_02 := pstMotorCtrl^; // Copia el contenido de la zona de memoria apuntada
a la
                        // estructura del motor 2, en este caso el resultado es el mismo
                        // que se obtendría con stMotor_02:= stMotor_01;

stMotor_03 := stMotor_02; // Copia los mismos valores al motor 3;

IF xMarcha THEN         // Si se pulsa marcha máquina
   pstMotorCtrl^.xMotorOn := TRUE; // Se activa el bit de marcha al que apunta el
puntero (stMotor_01).
   pstMotorCtrl^.xMotorOff := FALSE; // Se desactiva el bit de paro al que apunta el
puntero (stMotor_01)
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:

```
PROGRAM SR_Main_03
VAR
   aintFIFO    : ARRAY[1..20] OF INT; // Array de 20 enteros.
   aintFIFO2   : ARRAY[1..20] OF INT; // Array de 20 enteros.
   paint       : POINTER TO ARRAY[1..20] OF INT; // Puntero al array.
   pint        : POINTER TO INT; // Puntero a un entero.
END_VAR

// Ejemplo basico de como acceder a arrays mediante punteros:
paint := ADR(aintFIFO); // _Asignamos la dirección del array al puntero.
paint^[3] := 4; // Dentro del array podemos acceder a un elemento en concreto
aintFIFO2 := paint^; // O copiar el array apuntado entero, sobre otro array
pint := paint + (4 * SIZEOF (INT)); // Tambien se puede crear un puntero a un INT
para acceder a uno de los
                            // elementos del array. Tomamos la dirección inicial del array y
le
                            // sumampos un offeset de tantos bytes como se necesitan para
el tipo de
                            // datos INT y lo multiplicamos por el indice del array al que
queremos
                            // acceder. SIZEOF (TYPE) retorna el número de bytes según el
tipo de datos.

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

Pase por puntero:

```
// Función para calcular el area de un Rectangulo, con los valores contenidos en una
estructura de datos del tipo stRectangulo
// La estructura se pasa mediante un puntero a la estructura stRectangulo deseada y
la función retorna el resultado a la
// misma estructura.

FUNCTION Fc_AreaCalcPoint : REAL
VAR_INPUT
    i_ptstRect : POINTER TO st_Rectangulo; // Puntero de entrada con la dirección de
la estructura.
END_VAR

// El valor del area, de la estructura indicada por la dirección del puntero es igual al
// valor del lado A de la estructura indicada por la dirección del puntero por
// el valor del lado B de la estructura indicada por la dirección del puntero
i_ptstRect^.rArea := i_ptstRect^.rASide * i_ptstRect^.rBSide;
```

Pase por Referencia:

```
// Función para calcular el area de un Rectangulo, con los valores contenidos en una
estructura de datos del tipo stRectangulo
// La estructura se pasa por referencia.

FUNCTION Fc_AreaCalcRef : REAL
VAR_INPUT
    i_Ref : REFERENCE TO st_Rectangulo;
END_VAR

i_Ref.rArea := i_Ref.rASide * i_Ref.rBSide;
```

Ejemplo de código de llamadas a las funciones:

```
PROGRAM SR_Main_01
VAR
   inLocalAway : INT; // Variable integer local de SR_Main_01 para ser accedida externamente
   stRectangulo1 : st_Rectangulo; // Estructura que contiene los datos del rectangulo1 A, B y su area
   stRectangulo2 : st_Rectangulo; // Estructura que contiene los datos del rectangulo2 A, B y su area
   stRectangulo3 : st_Rectangulo; // Estructura que contiene los datos del rectangulo3 A, B y su area

   refRectangulo : REFERENCE TO st_Rectangulo := stRectangulo3; // Hace Referencia a stRectangulo3
END_VAR

// Asignación de valores a los lados de los tres rectángulos.

// Asignación de valores de los lados del rectángulo 1
stRectangulo1.rAside := 44; //Valor del lado A.
stRectangulo1.rBside := 32; //Valor del lado B.

// Asignación de valores de los lados del rectángulo 2
stRectangulo2.rAside := 12.8; //Valor del lado A.
stRectangulo2.rBside := 320.4; //Valor del lado B.

// Asignación de valores de los lados del rectángulo 3
stRectangulo3.rAside := 1024.2; //Valor del lado A.
stRectangulo3.rBside := 2048.4; //Valor del lado B.

// Cálculo del área del rectángulo pasando valores a la función
stRectangulo1.rArea := Fc_AreaCalcVal(i_rAside:=stRectangulo1.rAside, i_rBside:=stRectangulo1.rBside);

// Cálculo del área del rectángulo pasando un puntero a la función
Fc_AreaCalcPoint(ADR(stRectangulo2));

// Cálculo del área del rectángulo pasando una referencia a la función
Fc_AreaCalcRef(refRectangulo);
```

En este caso puede que 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:

```
1   // Esta función calcula la media de un buffer de 20 elementos. Solo a modo de
2   ejemplo comparativo
3   // no sería una forma muy adecuada de hacerlo así
4
5   FUNCTION Fc_AverageValues : REAL
6   VAR_INPUT
7      i_REALV1  : REAL; //Valor posición 1
8      i_REALV2  : REAL; //Valor posición 2
9      i_REALV3  : REAL; //Valor posición 3
10     i_REALV4  : REAL; //Valor posición 4
11     i_REALV5  : REAL; //Valor posición 5
12     i_REALV6  : REAL; //Valor posición 6
13     i_REALV7  : REAL; //Valor posición 7
14     i_REALV8  : REAL; //Valor posición 8
15     i_REALV9  : REAL; //Valor posición 9
16     i_REALV10 : REAL; //Valor posición 10
17     i_REALV11 : REAL; //Valor posición 11
18     i_REALV12 : REAL; //Valor posición 12
19     i_REALV13 : REAL; //Valor posición 13
20     i_REALV14 : REAL; //Valor posición 14
21     i_REALV15 : REAL; //Valor posición 15
22     i_REALV16 : REAL; //Valor posición 16
23     i_REALV17 : REAL; //Valor posición 17
24     i_REALV18 : REAL; //Valor posición 18
25     i_REALV19 : REAL; //Valor posición 19
26     i_REALV20 : REAL; //Valor posición 20
27  END_VAR
28
29  //Retorna la suma de todos los valores dividida del número de valores que son 20.
30
31  Fc_AverageValues := (i_REALV1 + i_REALV2 + i_REALV3 + i_REALV4 + i_REALV5 +
32  i_REALV6 + i_REALV7 +
33                 i_REALV8 + i_REALV9 + i_REALV10 + i_REALV11 + i_REALV12 +
    i_REALV13 +
                   i_REALV14 + i_REALV15 + i_REALV16 + i_REALV17 + i_REALV18 +
                   i_REALV19 + i_REALV20) / 20.0 ;
```

Código de la función Fc_AverageReferencia para pase por referencia:

```
1   // Esta función calcula la media de un buffer de 20 elementos. Solo a modo de
2   ejemplo comparativo
3   // pasando valores por referencia.
4
5   FUNCTION Fc_AverageReferencia : REAL
6
7   VAR_INPUT
8    i_Ref : REFERENCE TO ARRAY[1..20] OF REAL;
9   END_VAR
10  VAR
11   intIdx : INT;      // Variable indice para el bucle.
12   rVAcum : REAL:=0; // Valor acumulado.
13  END_VAR
14
15  // Retorna la suma de todos los valores divida del número de valores que son 20.
16
17  FOR intIdx:=1 TO 20 BY 1 DO
18   rVAcum := rVAcum + i_Ref[intIdx];
19  END_FOR;
    Fc_AverageReferencia := rVAcum / 20.0;
```

Código de ejemplo de llamada a ambas funciones:

```
PROGRAM SR_Main_04
VAR
 arFIFO  : ARRAY[1..20] OF REAL; // FIFO con los valores de fuerza registrados.
 intIdx  : INT;                  // Variable de indice.
 rIncAng : REAL;                 // Valor de incremento angular para generación de senoide.
 rValAng : REAL;                 // Valor actual de angulo.
 rValSin : REAL;                 // Amplitud de la senoide superpuesta.
 rVMed   : REAL;                 // Resultado del cálculo.

 refFIFO : REFERENCE TO ARRAY[1..20] OF REAL := arFIFO; // Crea una referencia y la asigna a arFIFO
 rMedRef : REAL;           // Resultado del cálculo para el ejemplo de pase de valores por Ref.
END_VAR

// Ejemplo de como realizar el cálculo del valor medio de las lecturas de fuerza contenidas en arFIFO
// mediante la función Fc_AverageValues (Pase de parámetros por valores) y Fc_AverageReferencia (Pase de
// parámetros por referencia)
// Lo que se pretende es ver las ventajas del pase por referencia

// Asignación de valores para llenar el FIFO a efectos de tener algunos valores para el cálculo de la media
// al valor 124 le superpone una variación senoidal de amplitud 6

rIncAng := (2 * 3.14159) / 20.0; // 2 * PI Radianes dividido entre 20 puntos.
rValAng := 0.0;                  // Valor inicial del angulo.

FOR intIdx :=1 TO 20 BY 1 DO
 rValSin := SIN(rValAng) * 6;         // Valor del seno para una amplitud de 6
 arFIFO[intIdx] := 124.0 + rValSin;  // A un nivel de 124.0 se superpone un seno de amplitud 6.
 rValAng := rValAng + rIncAng;       // Próximo valor angular.
END_FOR;

// Con el FIFO de valores llamaremos a la función para el cálculo de la media pasando valores.
// Lo que no sería para nada adecuado por tratarse de muchos parámetros.

rVMed:= Fc_AverageValues( i_REALV1  := arFIFO[1],
                          i_REALV2  := arFIFO[2],
                          i_REALV3  := arFIFO[3],
                          i_REALV4  := arFIFO[4],
                          i_REALV5  := arFIFO[5],
                          i_REALV6  := arFIFO[6],
                          i_REALV7  := arFIFO[7],
                          i_REALV8  := arFIFO[8],
                          i_REALV9  := arFIFO[9],
                          i_REALV10 := arFIFO[10],
                          i_REALV11 := arFIFO[11],
                          i_REALV12 := arFIFO[12],
                          i_REALV13 := arFIFO[13],
                          i_REALV14 := arFIFO[14],
                          i_REALV15 := arFIFO[15],
                          i_REALV16 := arFIFO[16],
```

```
                    i_REALV17 := arFIFO[17],
                    i_REALV18 := arFIFO[18],
                    i_REALV19 := arFIFO[19],
                    i_REALV20 := arFIFO[20]);

// Con el FIFO lleno de valores llamaremos a la función para el cálculo de la media pasando
valores por referencia
// para ver lo sencillo que resulta en este caso.

rMedRef := Fc_AverageReferencia(i_Ref:=refFIFO);
```

Claramente la llamada a la función pasando los valores por referencia es la mejor. Y en este ejemplo se ha supuesto un ejemplo con solo 20 datos de entrada, pero lo normal es encontrar aplicaciones con estructuras de datos de varios Kbytes.

- 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
- 🔗 AT&U, CODESYS - Difference between pointer and reference
- 🔗 AT&U, CODESYS -Differente between pass by vale and pass by Reference
- 🔗 Ninja Monkeys Tutorials, 21. TwinCAT 3: Pointer and References
- 🔗 Tutorial #26: Zeiger (Pointer) in CoDeSys

Link al Video de Youtube 011:

- 🔗 011 - OOP IEC 61131-3 PLC -- Puntero vs Referencia

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Palabra clave Abstracto

Palabra Clave Abstracto:

Concepto ABSTRACTO:

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.

Disponibilidad ABSTRACTO:

Ya estaba disponible en CODESYS, pero con el lanzamiento de TwinCAT 4024 ahora también está disponible en TwinCAT: la palabra clave ABSTRACT. (Disponible en TC3.1 Build 4024).

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.

La abstracción y el uso de la palabra clave abstract es una práctica común en OOP y muchos lenguajes de nivel superior como C# lo admiten. A menudo se considera como el cuarto pilar de la programación orientada a objetos.

¿Por qué necesitamos la abstracción?

Para comprender por qué la abstracción es tan importante en la programación orientada a objetos, volvamos rápidamente a la definición de abstracción. La abstracción consiste en ocultar al usuario detalles de implementación innecesarios y centrarse en la funcionalidad.

Considere un bloque de funciones que implementa una funcionalidad básica de celda de carga. Para usar esto, todo lo que necesitamos saber es que necesita una señal de entrada sin procesar y un factor de escala, y nos proporcionará un valor

de salida en Newton. No necesitamos saber cómo se convierte, filtra y escala el valor de salida. Deja que alguien más se preocupe por eso. No es de influencia en nuestro programa. Solo trabajaremos con una interfaz simple de una celda de carga.

Es bueno saber que el uso de abstracciones está estrechamente relacionado con el principio de inversión de dependencia, uno de los principios SOLID . Esto se vuelve especialmente importante cuando comienzas a trabajar con pruebas unitarias.

Reglas para el uso de la palabra clave ABSTRACT:

- Los bloques de funciones abstractas no se pueden instanciar.
- Los bloques de funciones abstractas pueden contener métodos y propiedades abstractos y no abstractos.
- Los métodos abstractos o las propiedades no contienen ninguna implementación (solo la declaración).
- Si un bloque de funciones contiene un método o propiedad abstracta, debe ser abstracta.
- Los bloques de funciones abstractas deben extenderse para poder implementar los métodos o propiedades abstractas. Por lo tanto: Un FB derivado debe implementar los métodos / propiedades de su FB básico o también debe definirse como abstracto. Muestra Clase básica abstracta:

  ```
  FUNCTION_BLOCK ABSTRACT FB_System_Base
  ```

  Los puntos en común de todos los módulos del sistema se implementan en esta clase básica abstracta. Contiene la propiedad no abstracta "nSystemID" y el método abstracto "Execute" para esto:

  ```
  PROPERTY nSystemID : UINT
  ```

  ```
  METHOD ABSTRACT Execute
  ```

  mientras que la implementación de "nSystemID" es la misma para todos los sistemas, la implementación del método "Execute" difiere para los sistemas individuales.

Subclase no abstracta:

```
FUNCTION_BLOCK FB_StackSystem EXTENDS FB_System_Base
```

Las clases no abstractas que se derivan de la clase básica se implementan para los sistemas específicos. Esta subclase representa una pila. Dado que no es abstracto, debe implementar el método "Execute" que define la ejecución específica de la pila:

```
METHOD Execute
```

Ejemplo de Demostracion de la palabra clave ABSTRACT en TwinCAT:

- 🔗 The ABSTRACT keyword, www.plccoder.com

Links de ABSTRACT:

- 🔗 ABSTRACT concept, infosys.beckhoff.com
- 🔗 The ABSTRACT keyword, www.plccoder.com
- 🔗 Tutorial #21: Einstieg in die objektorientierte Programmierung mit CoDeSys - Teil 4: Final/Abstract

Link al Video de Youtube 012:

- 🔗 012 - OOP IEC 61131-3 PLC -- Abstract

December 16, 2023 10:31:59

December 16, 2023 10:31:59

FB Abstracto vs Interface

FB Abstracto frente a Interface:

la diferencia entre utilizar un bloque de función abstracto y una interfaz es que el FB Abstracto es un tipo de plantilla que define un conjunto de variables y parámetros de entrada/salida para ser utilizados en diferentes partes del programa.

Por otro lado, una interfaz define un conjunto de métodos y atributos (propiedades) que deben ser implementados por cualquier clase que la implemente.

En resumen, los bloques de función abstractos son útiles cuando se necesita reutilizar código en diferentes partes del programa, mientras que las interfaces son útiles cuando se quiere asegurar que determinadas clases implementen ciertos métodos.

Imaginar que tienes un programa que controla diferentes tipos de motores, como motores eléctricos, motores a gasolina y motores diesel. Para crear una estructura modular y reutilizable, podrías crear un bloque de función abstracto llamado "Controlador de Motor" que tenga entradas para el tipo de motor, la velocidad y la dirección. Luego, este bloque de función abstracto puede ser utilizado en diferentes partes del programa para controlar los diferentes motores. El bloque de función abstracto define una plantilla común que se utiliza en diferentes partes del programa. Por otro lado, si quisieras asegurarte de que todas las clases que controlan motores implementen ciertos métodos (por ejemplo, un método para encender el motor y otro para apagarlo), podrías crear una interfaz llamada "Controlador de Motor" que defina estos métodos. Luego, cualquier clase que implemente esta interfaz deberá implementar estos métodos obligatoriamente. En resumen, los bloques de función abstractos son útiles cuando se necesita reutilizar código en diferentes partes del programa, mientras que las interfaces son útiles cuando se quiere asegurar que determinadas clases implementen ciertos métodos.

- Los bloques de funciones, los métodos y las propiedades se pueden marcar como abstractos. "desde TwinCAT V3.1 build 4024".
- Los FB abstractos solo se pueden usar como FB básicos para la herencia.
- La instanciación directa de FBs abstractos no es posible.Por lo tanto, los FB abstractos tienen cierta similitud con las interfaces.

Ahora, la pregunta es en qué caso se debe usar una interfaz y en qué caso un FB abstracto.

Métodos abstractos:

```
METHOD PUBLIC ABSTRACT DoSomething : LREAL
```

- Consisten exclusivamente en la declaración y no contienen ninguna implementación. El cuerpo del método está vacío.
- Puede ser público , protegido o interno . El modificador de acceso privado no está permitido.
- No puede ser declarada adicionalmente como definitiva.

Propiedades abstractas:

```
PROPERTY PUBLIC ABSTRACT nAnyValue : UINT
```

- Puede contener getters, setters o ambos.
- Getter y setter consisten solo en la declaración y no contienen ninguna implementación.
- Puede ser público , protegido o interno . El modificador de acceso privado no está permitido.
- No puede ser declarada adicionalmente como definitiva .

Bloques de funciones abstractas:

```
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).

Diferencias entre un FB abstracto y una interfaz:

Si un bloque de funciones consta exclusivamente de métodos abstractos y propiedades abstractas, entonces no contiene ninguna implementación y, por lo tanto, tiene cierta similitud con las interfaces. Sin embargo, hay algunas características especiales a considerar en detalle.

	Interfaz	**FB Abstracto**
admite herencia múltiple	+	-
puede contener variables locales	-	+
puede contener métodos no abstractos	-	+
puede contener propiedades no abstractas	-	+
admite más modificadores de acceso además de público	-	+
aplicable con matriz	+	solo através de PUNTERO

La tabla puede dar la impresión de que las interfaces pueden reemplazarse casi por completo por FB abstractos. Sin embargo, las interfaces ofrecen una mayor flexibilidad porque se pueden usar en diferentes jerarquías de herencia.

Por lo tanto, como desarrollador, desea saber cuándo se debe usar una interfaz y cuándo se debe usar un FB abstracto. La respuesta simple es preferiblemente ambos al mismo tiempo. Esto proporciona una implementación estándar en el FB base abstracto, lo que facilita su derivación. Sin embargo, cada desarrollador tiene la libertad de implementar la interfaz directamente.

Ejemplo:

Los bloques de funciones deben diseñarse para la gestión de datos de los empleados. Se hace una distinción entre empleados permanentes (FB_FullTimeEmployee) y empleados por contrato (FB_ContractEmployee). Cada empleado se identifica por su nombre (sFirstName), apellido (sLastName) y el número de personal (nPersonnelNumber). Las propiedades correspondientes se proporcionan para este propósito. Además, se requiere un método que genere el nombre completo, incluido el número de personal, como una cadena formateada (GetFullName()). El cálculo de los ingresos mensuales se realiza mediante el método GetMonthlySalary().

Lo resolveremos de 3 formas distintas:

- 1. Enfoque de solución: FB abstracto

- 2. Enfoque de solución: Interfaz

- 3. Enfoque de solución: combinación de FB abstracto e interfaz

Resumen, Conclusiones:

- Si el usuario no debe crear una instancia propia del FB (porque esto no parece ser útil), entonces los FB abstractos o las interfaces son útiles.
- Si se quiere tener la posibilidad de generalizar en más de un tipo básico, se debe utilizar una interfaz.
- Si se puede configurar un FB sin implementar métodos o propiedades, se debe preferir una interfaz a un FB abstracto.

links FB Abstracto frente a Interface:

- 🔗 FB abastracto frente a interfaz, stefanhenneken.net
- 🔗 The ABSTRACT keyword, www.plccoder.com
- 🔗 ABSTRACT concept, infosys.beckhoff.com

Link al Video de Youtube 013:

- 🔗 013 - OOP IEC 61131-3 PLC -- FB Abstract vs Interface

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Interface fluida

Interfaz Fluida:

Un diseño de programación popular en lenguajes de alto nivel como C# es el llamado 'código fluido' o 'interfaz fluida'. ¿qué es una interfaz fluida y cómo implementarla en texto estructurado? nos centraremos en una implementación de una interfaz fluida en texto estructurado.

¿Qué es una interfaz fluida?

Según wikipedia:

En ingeniería de software, una interfaz fluida es una API orientada a objetos cuyo diseño se basa en gran medida en el encadenamiento de métodos. Su objetivo es aumentar la legibilidad del código mediante la creación de un lenguaje específico de dominio (DSL). El término fue acuñado en 2005 por Eric Evans y Martin Fowler.

Un buen ejemplo de este 'encadenamiento de métodos' se puede ver con las declaraciones LINQ de C#:

```
EmployeeNames = EmployeeList.Where(x=› x.Age › 65)
                .Select(x=› x)
                .Where(x=› x.YearsOfEmployment › 20)
                .Select(x=› x.FullName);
```

Al encadenar continuamente los métodos, podemos construir nuestra declaración completa. ¡Es bueno saber que una interfaz fluida se usa a menudo junto con un patrón de construcción!. Podemos pensar en la interfaz fluida como un concepto, mientras que el encadenamiento de métodos es una implementación. El objetivo del diseño fluido de la interfaz es poder aplicar múltiples propiedades a un objeto conectando los métodos con puntos **(.)** en lugar de tener que aplicar cada método individualmente.

¿Por qué queremos la Interfaz Fluida?

- Por legilibilidad, mas legible.
- Mas simple.
- Por mantenimiento.
- Por claridad.

- Por facilidad de escribir.
- Fácil de extender.

¿Cómo construimos una interfaz fluida?

Al hacer que el código sea comprensible y fluido, la interfaz fluida le da la impresión de que está leyendo una oración. Para lograr este patrón de diseño, necesitaría usar **el encadenamiento de métodos**.

En esta técnica, cada método devuelve un objeto y puede encadenar todos los métodos.

- veanse los links a los que se hace referencia, veremos un ejemplo en el cual implementaremos una interface fluida para realizar operaciones matematicas...

Links Interface Fluida:

- 🔗 fluent-code, www.plccoder.com
- 🔗 fluent-interface-and-method-chaining-in-twincat-3
- 🔗 tc3-data-logger creado con interface fluida, github.com/benhar-dev
- 🔗 interface fluida por referencia, getting-limits-twincat-ralph-koettlitz

Link al Video de Youtube 014:

- 🔗 014 - OOP IEC 61131-3 PLC -- Interface Fluida

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Interface vs Herencia

Interface vs Herencia:

Herencia:

- Debemos definir la implementación de la clase base.
- Las clases heredadas dependen de la clase base.
- La jerarquia de herencia profunda produce alta dependencia, y esto no esta bien lo que se busca es una baja dependencia y alta cohesión.
- La jerarquia de herencia profunda puede complicarse si es necesario cambiar la clase base.
- La jerarquia de herencia profunda por regla general no deberia pasar de más de 3 niveles de herencia.
- Administrar el acceso a datos con especificadores de acceso puede ser más difícil con una gran herencia.
- La herencia múltiple en una misma Clase no es compatible.

Interface:

- La clase base (clase abstracta) no tiene implementación.
- No hay dependencias entre las clases que implementan la misma interfaz.
- Se permite la implementación de múltiples interfaces en una misma Clase.

Las Interfaces y la Herencia pueden trabajar de la mano, utilizarse a la vez cogiendo de cada una lo mejor posible:

Links Interface vs Herencia:

- 🔗 www.techandsolve.com,alta-cohesion-y-bajo-acoplamiento-en-diseno-de-software
- 🔗 object-oriented-programming-plc-using-inheritance-ralph-koettlitz
- 🔗 stefanhenneken.net,iec-61131-3-object-composition-with-the-help-of-interfaces

Links Videos de Youtube 015 y 016:

- 🔗 015 - OOP IEC 61131-3 PLC -- Interface vs Herencia_1
- 🔗 016 - OOP IEC 61131-3 PLC -- Interface vs Herencia_2

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Otros Operadores

Otros Operadores:

__DELETE:

- El operador es una extensión del estándar IEC 61131-3.
- El operador libera la memoria de instancias, que el operador __NEW generó dinámicamente.
- El operador DELETE no tiene valor de retorno y el operando se establece en 0 después de esta operación.

```
//Syntax:
 __DELETE (<Pointer>)
```

- Si un puntero apunta a un bloque de funciones, TwinCAT llama al método correspondiente FB_exit antes de que el puntero se establezca en 0.
- 🔗 __DELETE, infosys.beckhoff.com
- 🔗 __DELETE, help.codesys.com

__ISVALIDREF:

- El operador es una extensión del estándar IEC 61131-3.
- El operador se utiliza para comprobar si una referencia apunta a un valor. Por lo tanto, la verificación es comparable con una verificación de 'desigual a 0' en el caso de una variable de puntero.
- Puede encontrar una descripción de la aplicación y una muestra del uso del operador en la descripción del tipo de datos REFERENCE.
- El operador __ISVALIDREF solo se puede utilizar para operandos de tipo REFERENCE TO. Este operador no se puede utilizar para comprobar las variables de la interfaz. Para verificar si a una variable de interfaz ya se le asignó una instancia de bloque de funciones, puede verificar que la variable de interfaz no sea igual a 0 (IF iSample <> 0 THEN ...).

```
//Syntax:
<Boolean variable> := __ISVALIDREF(<with REFERENCE TO <data type>
declared identifier);
```

- 🔗 __ISVALIDREF, infosys.beckhoff.com
- 🔗 __ISVALIDREF, help.codesys.com

__NEW:

- El operador es una extensión del estándar IEC 61131-3.
- El operador __NEW asigna memoria para instancias de bloques de funciones o matrices de tipos de datos estándar. El operador devuelve un puntero con tipo adecuado al objeto. Si no utiliza el operador dentro de una asignación, TwinCAT emite un mensaje de error.
- Si falla el intento de asignación de memoria, __NEW devuelve el valor 0.
- La memoria dinámica se asigna desde el grupo de memoria del enrutador.
- Información de estado del enrutador TwinCAT : El bloque de funciones FB_GetRouterStatusInfo de la biblioteca Tc2_Utilities se puede utilizar para leer información de estado del enrutador TwinCAT, como la memoria del enrutador disponible, desde el PLC.

```
//Syntax:
 __NEW (<Type>,[<Length>])
```

- Ningún cambio de tipo es posible a través del cambio en línea
- Un módulo de función/DUT, que se puede crear con __NEW, ocupa un área de memoria fija. No puede cambiar su diseño de datos utilizando la función de cambio en línea. Esto significa que no se pueden agregar nuevas variables, no se pueden eliminar variables y no se pueden cambiar tipos. Esto garantiza que el puntero a este objeto siga siendo válido después del cambio en línea.
- Por esta razón, el operador __NEW solo se puede aplicar a bloques de función/DUT de bibliotecas y a bloques de función/DUT con el **{attribute 'enable_dynamic_creation'}**. Si se modifica la interfaz de un bloque de función/DUT de este tipo, TwinCAT emite un mensaje de error.
- 🔗 __NEW, infosys.beckhoff
- 🔗 __NEW, help.codesys.com

__QUERYINTERFACE:

- El operador es una extensión del estándar IEC 61131-3.

- En tiempo de conversión, el operador realiza un tipo de referencia de interfaz a otra. El operador devuelve un resultado de tipo BOOL. TRUE significa que TwinCAT realizó la conversión con éxito.

```
//Syntax:
__QUERYINTERFACE(<ITF_Source>,<ITF_Dest>)
```

- 1er operando: referencia de interfaz o instancia de FB.
- 2.º operando: referencia de interfaz con los tipos de destino requeridos.El segundo parámetro contiene una referencia a la interfaz solicitada.

Un requisito previo para la conversión explícita es que tanto ITF_Source como ITF_Dest sean derivados de __System.IQueryInterface. Esta interfaz está implícitamente disponible y no requiere biblioteca.

- 🔗 __QUERYINTERFACE, infosys.beckhoff.com
- 🔗 __QUERYINTERFACE, help.codesys.com

__QUERYPOINTER:

- El operador es una extensión de IEC61131-3.
- El operador habilita la conversión de tipo de una referencia de interfaz de un bloque de funciones a un puntero en tiempo de ejecución. El operador devuelve un resultado de tipo BOOL. TRUE significa que TwinCAT realizó la conversión con éxito.
- Por razones de compatibilidad, la definición del puntero a convertir debe ser una extensión de la interfaz básica __SYSTEM.IQueryInterface.

```
//Syntax:
__QUERYPOINTER (<ITF_Source>, <Pointer_Dest>)
```

- El primer operando asignado al operador es una referencia de interfaz o una instancia de FB con los tipos de destino deseados, el segundo operando es un puntero. Después de procesar __QUERYPOINTER, Pointer_Dest contiene el puntero a la referencia o instancia de un bloque de funciones, al que apunta actualmente la referencia de interfaz ITF_Source. Pointer_Dest no tiene tipo y se puede convertir a cualquier tipo. Asegúrese de que el tipo sea correcto. Por ejemplo, la interfaz podría ofrecer un método que devuelva un código de tipo.
- 🔗 __QUERYPOINTER, infosys.beckhoff.com
- 🔗 __QUERYPOINTER, help.codesys.com

__TRY, __CATCH, __FINALLY, __ENDTRY:

- Los operadores son una extensión del estándar IEC 61131-3 y se utilizan para un manejo de excepciones específico en el código IEC.
- Disponible desde TC3.1 compilación 4024 para sistemas de tiempo de ejecución de 32 bits.
- Ahora también está disponible para sistemas de destino de 64 bits a partir de TwinCAT versión 3.1.4026

```
//Syntax:
__TRY
    <try_statements>

__CATCH(exc)
    <catch_statements>

__FINALLY
    <finally_statements>

__ENDTRY

<further_statements>
```

- Si una instrucción que aparece bajo el operador __Try genera una excepción, el programa del PLC no se detiene. En su lugar, ejecuta las instrucciones bajo _CATCH y, por lo tanto, inicia el manejo de excepciones. A continuación, se ejecutan las instrucciones bajo __FINALLY. El manejo de excepciones termina con __ENDTRY. A continuación, el programa de PLC ejecuta las instrucciones posteriores (instrucciones después de __ENDTRY).
- Las instrucciones del bloque _TRY, que se encuentran debajo de la instrucción que desencadena la excepción, ya no se ejecutan. Esto significa que tan pronto como se descarta la excepción, se cancela la ejecución posterior del bloque _TRY y se ejecutan las instrucciones bajo _CATCH.
- Las instrucciones bajo _FINALLY siempre se ejecutan, es decir, incluso si las instrucciones bajo _TRY no lanzan ninguna excepción.
- Una variable IEC para una excepción tiene el tipo de datos __SYSTEM.ExceptionCode.
- 🔗 __TRY, __CATCH, __FINALLY, __ENDTRY, help.codesys.com
- 🔗 __TRY, __CATCH, __FINALLY, __ENDTRY, infosys.beckhoff.com

__VARINFO:

- El operador es una extensión del estándar IEC 61131-3.
- El operador devuelve información sobre una variable. Puede guardar la información como una estructura de datos en una variable de tipo de datos __SYSTEM.VAR_INFO.

```
//Syntax in the declaration:
<name of the info variable> : __SYSTEM.VAR_INFO; // Data structure for info variable

//Syntax for the call:
<name of the info variable> := __VARINFO( <variable name> ); // Call of the operator
```

```
//Sample:
//En tiempo de ejecución, la variable MyVarInfo contiene la información sobre la variable nVar.

VAR
    MyVarInfo : __SYSTEM.VAR_INFO;
    nVar      : INT;
END_VAR
MyVarInfo := __VARINFO(nVar);
```

- 🔗 __VARINFO, infosys.beckhoff.com
- 🔗 __VARINFO, help.codesys.com

TEST_AND_SET:

- Puede usar esta función para verificar y establecer una bandera. No hay opción para interrumpir el proceso. Esto permite sincronizar los accesos a los datos. El modo de operación de un semáforo se puede lograr con TestAndSet.
- Si la llamada a la función tiene éxito, la función devuelve VERDADERO y se puede acceder a los datos deseados. Si la llamada a la función no tiene éxito, la función devuelve FALSO y no se puede acceder a los datos deseados. En este caso, se debe prever un tratamiento alternativo.

🔗 TEST_AND_SET, Codesys

🔗 TESTANDSET, TwinCAT

- El bloque de funciones FB_IecCriticalSection ofrece la aplicación de secciones críticas como método Mutex alternativo.

Links Otros Operadores:

- 🔗 Further operators, infosys.beckhoff.com
- 🔗 Others Operators, help.codesys.com
- 🔗 stefanhenneken.net,iec-61131-3-object-composition-with-the-help-of-interfaces
- 🔗 20. TwinCAT 3: Structures: Alignment & dynamically created structures

Links Videos de Youtube 016, 017 y 018:

- 🔗 016 - OOP IEC 61131-3 PLC -- Interface vs Herencia_2
- 🔗 017 - OOP IEC 61131-3 PLC -- Otros Operadores1
- 🔗 018 - OOP IEC 61131-3 PLC -- Otros Operadores2

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Texto Estructurado Extendido

ExST - Texto Estructurado Extendido:

- El Texto Estructurado Extendido (ExST) es una extensión específica de CODESYS, que la empresa 3S ha implementado en CODESYS.
- Además del lenguaje básico de IEC 61131-3, se han agregado algunos elementos de control interesantes, tan útiles para el uso diario que me gustaría presentárselos directamente.
- También tenemos ExST disponible para TwinCAT3.

Asignación Extendida como Expresión:

- En ExST, como extensión del estándar IEC 61131-3, CODESYS, TwinCAT permite el uso de asignaciones como expresiones.

Ejemplo:

```
myVar := myVar1 := myVar2 + 26 ; // asignación extendida
```

- En este ejemplo, las variables myVar y myVar1 reciben el valor de la variable myVar2 sumado a 26.

Tabla de Ejemplos:

Ejemplo	**Comentario**
int_var1 := int_var2 := int_var3 + 9;	(*int_var1 y int_var2 recibe el valor de int_var3 + 9*)
real_var1 := real_var2 := int_var;	(*real_var1 y real_var2 recibe el valor de int_var*)
int_var := real_var1 := int_var;	(*asignación incorrecta, ¡los tipos de datos no corresponden!*)

Ejemplo	Comentario
IF b := (i = 1) THEN i := i + 1; END_IF	

- Hasta ahora, la asignación siempre se ha hecho con "**:=**" El valor del lado derecho se asigna a la variable del lado izquierdo.

S=

- Sin embargo, también puede realizar la asignación utilizando "**S=**".
- Este operador establece una variable.
- Cuando el valor de la variable a la izquierda de "**S=**" se convierte en TRUE una vez, sigue siendo TRUE, incluso si el operando a la derecha de "**S=**" vuelve a FALSE.

Sintaxis:

```
<variable1> S= <variable2>;
```

Ejemplo:

```
bVar1 S= bVar2;
```

- bVar1 obtiene el valor de bVar2. Una vez que bVar1 se ha establecido en TRUE, bVar1 permanece TRUE aunque vuelva bVar2 a ser FALSE.

R=

- El operador de asignación "**R=**" es el opuesto de "**S=**".
- Esta asignación restablece una variable.
- Una vez que el valor de la variable a la izquierda de "**R=**" se ha convertido en FALSE, permanece FALSE incluso si el operando a la derecha de "**R=**" vuelve a cambiar a TRUE .

Sintaxis:

```
<variable1> R= <variable2>;
```

Ejemplo:

```
bVar11 R= bVar22;
```

- bVar11 obtiene el valor de bVar22. Una vez que bVar11 se ha establecido en FALSE, bVar11 permanece FALSE aunque vuelva bVar22 a ser TRUE.

El uso de "**S=**" y "**R=**" en una concatenación es interesante. Si aplica el operador Set a una variable y el operador Reset a otra en una línea, la referencia siempre es al último elemento de la serie de asignación.

Ejemplo:

```
bVar1 S= bVar2 R= F_Fun1(nPar1, nPar2);
```

- Todas las asignaciones de configuración y restablecimiento siempre hacen referencia al último elemento de la asignación. Ejemplo: En este caso, bVar2 obtiene la salida de F_Fun1 resultante del reinicio, pero: bVar1 no obtiene el resultado del conjunto de bVar2, sino el resultado del conjunto de F_Fun1!

```
bMyVar1 S= bMyVar2 R= myTimer.Q;
```

- Aquí bMyVar2 se restablece cuando el tiempo establecido en el módulo temporizador ha expirado.
- **¡¡Atención!!** La variable bMyVar1 ahora no se establece si bMyVar2 tiene el valor TRUE, pero los operadores de asignación siempre actúan en el bloque del temporizador.

Links ExST:

- 🔗 Threes Soup01, Beckhoff.Tutorial About TwinCAT3 ExST
- 🔗 Threes Soup01, Beckhoff.TwinCAT ExST
- 🔗 Structured Text and Extended Structured Text (ExST), infosys.beckhoff.com
- 🔗 infosys.beckhoff.com, ExST
- 🔗 help.codesys.com, ExST
- 🔗 www.codesys-blog.com, ExST
- 🔗 Texto estructurado (ST), Texto estructurado extendido (ExST)

Links Video de Youtube 019 :

- 🔗 019 - OOP IEC 61131-3 PLC -- ExST Texto Estructurado Extendido

December 16, 2023 10:31:59

December 16, 2023 10:31:59

4 Pilares

Principios OOP: (4 pilares)

- Abstracción -- La forma de **plasmar 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 caracteristicas** 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

Links de Principios OOP:

- 🔗 github.com/Aliazzzz/OOP-Concept-Examples-in-CODESYS-V3
- 🔗 4 PRINCIPIOS de la PROGRAMACIÓN ORIENTADA A OBJETOS
- 🔗 Programación Orientada a Objetos (POO): Abstracción, Encapsulamiento, Herencia, Polimorfismo

Link al Video de Youtube 020:

- 🔗 020 - OOP IEC 61131-3 PLC -- 4 Pilares OOP - Abstracción

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Abstracción

ABSTRACCION:

La Abstracción es el proceso de ocultar información importante, mostrando solo lo información más esencial. Reduce la complejidad del código y aísla el impacto de los cambios. La abstracción se puede entender a partir de un ejemplo de la vida real: encender un televisor solo debe requieren hacer clic en un botón, ya que las personas no necesitan saber el proceso por el que pasa. Aunque ese proceso puede ser complejo e importante, no es necesario que el usuario sepa cómo se implementa. La información importante que no se requiere está oculta para el usuario, reduciendo la complejidad del código, mejorando la ocultación de datos y la reutilización, haciendo así que los Bloques de Funciones sean más fáciles de implementar y modificar.

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.

Links Abstracción:

- 🔗 ABSTRACT,www.plccoder.com
- 🔗 ABSTRACION Concepto, Infosys Beckhoff

Link al Video de Youtube 020:

- 🔗 020 - OOP IEC 61131-3 PLC -- 4 Pilares OOP - Abstracción

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Encapsulamiento

Encapsulamiento:

¿Qué es la Encapsulación?

- La encapsulación agrupa propiedades y métodos en un sola clase (bloque de funciones).
- La encapsulación se utiliza para agrupar datos con los métodos que operan en ellos y para ocultar datos en su interior, evitando que personas no autorizadas accedan directamente a la clase.
- Con encapsulación también nos referimos a la capacidad de un bloque de funciones para ocultar datos y comportamientos que no son necesarios para el usuario.
- Es decir, hacemos una separación entre una interfaz de bloques de funciones y su implementación.
- 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.

¿Por qué es importante la encapsulación?

- Podemos especificar la accesibilidad de los miembros de un bloque de funciones.
- Ayuda a proteger sus datos de la corrupción accidental.
- Ayuda a mantener su código limpio y extensible.

¿Cómo logramos la encapsulación?

En Codesys y TwinCAT podemos usar un bloque de funciones para construir el proyecto original de un objeto (como una clase en C#). Con la ayuda de las propiedades y los métodos podemos hacer 'puertos de entrada' a nuestros campos y funcionalidades internas.

Conclusión:

La Encapsulación es uno de los 4 pilares de OOP. La encapsulación consiste en agrupar métodos y propiedades en un bloque de funciones y ocultar y proteger datos que no son necesarios para el usuario. Esto nos ayuda a escribir código SÓLIDO y reutilizable.

Links Encapsulacion:

- 🔗 www.plccoder.com, Encapsulation
- 🔗 es.wikipedia.org, Encapsulamiento

Link al Video de Youtube 021:

- 🔗 021 - OOP IEC 61131-3 PLC -- 4 Pilares OOP - Encapsulamiento y Herencia

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Herencia

Herencia:

- La herencia permite al usuario crear clases basadas en otras clases.
- Las clases heredadas pueden utilizar las funcionalidades de la clase base, así como algunas funcionalidades adicionales que el usuario puede definir.
- Elimina el código redundante, evita copiar y pegar y facilita la expansión.
- Esto es muy útil porque permite ampliar o modificar (anular) las clases sin cambiar la implementación del código de la clase base.

Ejemplo de Herencia:

¿Qué tienen en común un teléfono fijo antiguo y un smartphone?

- Ambos pueden ser clasificados como teléfonos.

¿Deberían clasificarse como objetos?

- No, ya que también definen las propiedades y comportamientos de un grupo de objetos. Un teléfono inteligente funciona como un teléfono normal, pero también es capaz de tomar fotografías, navegar por Internet y hacer muchas otras cosas. Entonces, teléfono fijo antiguo y el teléfono inteligente son clases secundarias que amplían la clase de teléfono principal.

Definiciones de Herencia:

- Superclase: La clase cuyas características se heredan se conoce como **superclase** (ó una clase **base** ó una clase **principal** ó clase **padre**).
- Subclase: La clase que hereda la otra clase se conoce como **subclase** (ó una clase **derivada**, clase **extendida** ó clase **hija**).

Links Herencia:

- 🔗 stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance

Link al Video de Youtube 021:

- 🔗 021 - OOP IEC 61131-3 PLC -- 4 Pilares OOP - Encapsulamiento y Herencia

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Polimorfismo

Polimorfismo:

El concepto de polimorfismo se deriva de la combinación de dos palabras: Poly (Muchos) y Morfismo (Forma). Refactoriza casos de cambio/declaraciones de casos feos y complejos. El polimorfismo permite que un objeto cambie su apariencia y desempeño dependiendo de la situación práctica para poder realizar una determinada tarea. Puede ser estático o dinámico:

- El polimorfismo estático ocurre cuando el compilador define el tipo de objeto;
- El polimorfismo dinámico se produce cuando el tipo se determina durante el tiempo de ejecución, lo que hace posible para que una misma variable acceda a diferentes objetos mientras el programa se está ejecutando.

Ejemplos de Polimorfismo:

- Un buen ejemplo para explicar el polimorfismo es una navaja suiza. Una navaja suiza es una herramienta única que incluye un montón de recursos que se pueden utilizar para resolver problemas diferentes. Al seleccionar la herramienta adecuada, se puede utilizar una navaja suiza para realizar un determinado conjunto de tareas valiosas.
- De la manera dual, otro ejemplo podria ser un bloque sumador simple que se adapta para hacer frente a, por ejemplo, los tipos de datos int, float, string y time es un ejemplo de un polimórfico recurso de programación.

¿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).

Tambien se puede conseguir el Polimorfismo por Referencia y/o por Punteros:

- Referencia: (REFERENCE)
- Puntero: (POINTER)

Tipos de Polimorfismo:

- Paramétrico:
  - El Polimorfismo **Paramétrico** va ocurrir cuando definamos en una clase varios metodos que van a tener el mismo nombre pero diferentes parametros. La cantidad y tipos de parametros es la diferencia.
- Sobrecarga:
  - El Polimorfismo por **Sobrecarga** va a haber un comportamiento diferente dependiendo de los parametros que se han recibido. Lo vamos a encontrar cuando distintas clases contienen metodos con el mismo nombre pero con un comportamiento diferente.
- Como Novedad En TwinCAT build 3.1.4026, y en Codesys 3.5 SP16 se pueden declarar parametros de entrada con inicialización con esto no sera obligario al llamar al objeto que instancia dicha clase poner dicho pametro de entrada en la llamada se puede obviar, con esto no conseguimos totalmente un polimorfismo paramétrico ya que no podemos tener el mismo nombre de metodo pero con diferentes parametros, pero al menos es un pequeño avance...

Links Polimorfismo:

- 🔗 polymorphism, www.plccoder.com
- 🔗 abstract, www.plccoder.com
- 🔗 stefanhenneken.net,iec-61131-3-methods-properties-and-inheritance
- 🔗 AT&U, CODESYS - Runtime polymorphism using inheritance (OOP)
- 🔗 AT&U,CODESYS - Runtime polymorphism using an ITF (OOP)

Link al Video de Youtube 022:

- 🔗 022 - OOP IEC 61131-3 PLC -- 4 Pilares OOP - Polimorfismo

December 16, 2023 10:31:59

December 16, 2023 10:31:59

SOLID

- 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**.

Los principios SOLID son una parte esencial del desarrollo de software orientado a objetos y han demostrado ser herramientas valiosas para desarrollar código limpio, mantenible y extensible. En la tecnología de automatización industrial,

especialmente en la programación de controladores con IEC 61131-3, es de particular importancia desarrollar sistemas robustos y confiables.

Además de los principios SOLID, existen otros principios como:

Keep It Simple, Stupid (KISS).

```
" Mantenlo Simple, Estúpido "
```

- Evite la complejidad innecesaria en su código, use soluciones simples para resolver problemas.

- **Ejemplo:** En lugar de escribir un algoritmo personalizado para generar un número aleatorio dentro de un rango, use el generador de números aleatorios incorporado en su lenguaje de programación.

Don't Repeat Yourself (DRY).

```
" No te repitas "
```

- Cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema.
- Evite la duplicación de código y mantenga su base de código lo más mantenible y escalable posible.
- **Ejemplo:** En lugar de copiar y pegar el mismo bloque de código en varios lugares, cree una función o módulo que se pueda reutilizar.

Law Of Demeter (LOD).

```
" Habla Solo con tus amigos inmediatos "
```

- La Ley de Demeter (LOD) en programación es un principio que establece que un objeto debe tener acceso limitado a los objetos relacionados con él y solo interactuar con los objetos más cercanos a él. En resumen, un objeto no debe conocer la estructura interna de otros objetos y solo debe comunicarse con ellos a través de una interfaz limitada.
- **Ejemplo:** Si tienes una clase "Persona" que tiene un método "getNombre()" y otra clase "Empresa" que tiene un método "getPersona()". En lugar de acceder directamente al nombre de la persona desde la clase Empresa, se debería llamar al método "getNombre()" de la clase Persona desde fuera de la clase Empresa, para evitar una dependencia innecesaria y mantener una comunicación limitada entre objetos.

You Ain't Gonna Need It (YAGNI).

```
" No lo vas a necesitar "
```

- No agregue funcionalidad a su código hasta que realmente lo necesite.
- **Ejemplo:** No agregue una función a su aplicación que permita a los usuarios cambiar el color de la fuente si no es parte de los requisitos principales.

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

Los principios SOLID no son reglas o leyes que deban seguirse estrictamente. Son pautas que pueden ayudarnos a mejorar nuestra calidad de código y habilidades de diseño. No están destinados a ser aplicados ciega o dogmáticamente. Están destinados a ser utilizados con sentido común y juicio.

Links SOLID:

- 🔗 Cómo explicar conceptos de programación orientada a objetos a un niño de 6 años
- 🔗 iec-61131-3-solid-five-principles-for-better-software,stefanhenneken.net
- 🔗 Libro SOLID IEC61131-3 en Aleman de Stefanhenneken
- 🔗 kentcdodds.com,aha-programming
- 🔗 Qué son los principios SOLID ? Por qué son tan importantes y verás que ya los aplicas sin saberlo !!
- 🔗 Los Principios SOLID explicados ¡Con ejemplos! 100% PRÁCTICO
- 🔗 Cómo implementar los principios SOLID en JAVA
- 🔗 Principios de programación SOLID (#Shorts)

Link al Video de Youtube 023:

- 🔗 023 - OOP IEC 61131-3 PLC -- SOLID

December 16, 2023 10:31:59

December 16, 2023 10:31:59

SRP - Principio de Responsabilidad Única

Principio de Responsabilidad Única -- (Single Responsibility Principle) SRP :

El principio de responsabilidad única establece que una clase debe tener una sola responsabilidad en un programa.

Ejemplo :

Por ejemplo, en lugar de tener una clase "Empleado" que maneje tanto la información personal como el registro de tiempo, se deben crear dos clases separadas: "Empleado" para la información personal y "RegistroDeTiempo" para el registro de tiempo. De esta manera, cada clase se enfoca en una sola tarea y es más fácil de mantener y modificar.

En lugar de tener una Clase que maneje todo, creamos dos Clases separadas:

```
FUNCTION_BLOCK Empleado
VAR_INPUT
    Nombre : STRING;
    Apellido : STRING;
    CorreoElectronico : STRING;
END_VAR

// constructor
Empleado(ST_Empleado);

// getters y setters
nombre();
apellido();
correoElectronico();

END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK RegistroDeTiempo
VAR_INPUT
    empleado : ST_Empleado; // instancia de la función Empleado
    horaEntrada : DATE_AND_TIME;
    horaSalida : DATE_AND_TIME;
END_VAR

// constructor
RegistroDeTiempo(ST_RegistroDeTiempoEmpleado);

// getters y setters
empleado();
horaEntrada();
horaSalida();

END_FUNCTION_BLOCK
```


De esta manera, la Clase "Empleado" solo maneja la información personal del empleado y la Clase "RegistroDeTiempo" solo maneja el registro de tiempo. Cada Clase tiene una sola responsabilidad y es más fácil de mantener y modificar en el futuro.

Links:

- 🔗 stefanhenneken.net,iec-61131-3-solid-the-single-responsibility-principle
- 🔗 hdeleon.net, Principios SOLID: El Principio de Responsabilidad Única SRP
- 🔗 El Camino Dev, Principio de Responsabilidad Única en C# | Principios SOLID
- 🔗 makigas: aprende a programar, SOLID: Principio de Responsabilidad Única (SRP)

- 🔗 tech.tribalyte.eu, blog-solid-single-responsability

Link al Video de Youtube 024:

- 🔗 024 - OOP IEC 61131-3 PLC -- SOLID - SRP

December 16, 2023 10:31:59

December 16, 2023 10:31:59

OCP - Principio de Abierto/Cerrado

Principio de Abierto/Cerrado -- (Open/Closed Principle) OCP :

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, etc...

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.

Ejemplo:

```
FUNCTION_BLOCK Vehiculo
VAR_INPUT
    velocidad : REAL;
END_VAR

// método para obtener la velocidad
getVelocidad() : REAL;
END_FUNCTION_BLOCK

FUNCTION_BLOCK Coche EXTENDS Vehiculo // extiende la función Vehiculo
VAR_INPUT
    velocidadMaxima : REAL;
END_VAR

// método para obtener la velocidad máxima
getVelocidadMaxima() : REAL;
END_FUNCTION_BLOCK

FUNCTION_BLOCK Moto EXTENDS Vehiculo // extiende la función Vehiculo
VAR_INPUT
    aceleracion : REAL;
END_VAR

// método para obtener la aceleración
getAceleracion() : REAL;
END_FUNCTION_BLOCK
```

De esta manera, la clase "Vehiculo" está cerrada para modificaciones directas y abierta para extensiones a través de las nuevas clases "Coche" y "Moto". Cada nueva clase agrega funcionalidades específicas sin modificar directamente la clase original.

Links:

- 🔗 stefanhenneken.net, EC 61131-3: SOLID – The Open/Closed Principle
- 🔗 hdeleon.net, Principios SOLID: Principio de Abierto/Cerrado OCP
- 🔗 El Camino Dev, El Principio Abierto Cerrado con C# | Principios SOLID
- 🔗 makigas: aprende a programar, SOLID: Principio Abierto-Cerrado (OCP)
- 🔗 tech.tribalyte.eu, blog-solid-open-closed

Link al Video de Youtube 025:

- 🔗 025 - OOP IEC 61131-3 PLC -- SOLID - OCP

December 16, 2023 10:31:59

December 16, 2023 10:31:59

LSP - Principio de Sustitución de Liskov

Principio de Sustitución de Liskov -- (Liskov Substitution Principle) LSP :

- Este principio de la programación orientada a objetos debe su nombre a Barbara Liskov, reconocida ingeniera de software que fue la primera mujer de los Estados Unidos en conseguir un doctorado en Ciencias de la Computación, ganadora de un premio Turing y nombrada doctora honoris causa por la UPM.
- El principio de sustitución de Liskov establece que una instancia de una subclase debe poder ser utilizada en cualquier lugar donde se espera una instancia de la clase base, sin afectar el comportamiento del programa.

Ejemplo:

```
INTERFACE I_Mover
   METHODS Mover : REAL; // método para mover el vehículo
      VAR_INPUT
      Velocidad_Deseada : REAL;
      END_VAR
END_INTERFACE

FUNCTION_BLOCK Vehiculo IMPLEMENTS I_Mover
// clase base para los vehículos
VAR_INPUT
   velocidad : REAL;
END_VAR
   METHODS Mover : REAL; // método para mover el vehículo
      Mover := Velocidad_Deseada;
END_FUNCTION_BLOCK

FUNCTION_BLOCK Coche1 EXTENDS Vehiculo
// subclase para los coches
VAR_INPUT
   velocidadMaxima : REAL;
END_VAR
  METHODS Mover : REAL; // método para mover el Coche
   Mover := Velocidad_Deseada * velocidadMaxima;
END_FUNCTION_BLOCK

FUNCTION_BLOCK Moto1 EXTENDS Vehiculo
// subclase para las motos
VAR_INPUT
   Aceleracion : REAL;
END_VAR
   METHODS Mover : REAL; // método para mover la Moto
      Mover := Velocidad_Deseada * Aceleracion;
END_FUNCTION_BLOCK

FUNCTION_BLOCK Conductor
VAR_INPUT
   vehiculo : REFERENCE TO Vehiculo; // referencia a la clase base Vehiculo
END_VAR
   METHODS Mover : REAL; // método para mover el vehículo
      // método para mover el vehículo a la velocidad especificada
      THIS^.vehiculo.Mover(velocidad);
END_FUNCTION_BLOCK
```

- En este ejemplo, se utiliza la subclase `Coche1` y `Moto1` como instancias de la clase base `Vehiculo`, lo que cumple con el principio de sustitución de Liskov. Esto significa que se puede utilizar cualquier instancia de `Coche1` o `Moto1` donde se espera una instancia de `Vehiculo`, sin afectar el comportamiento del programa.
- Además, cada subclase tiene un método `Mover` que se utiliza para mover el vehículo, lo que demuestra cómo se puede utilizar la misma interfaz `I_Mover` (el mismo nombre de método) para diferentes implementaciones concretas, mediante la sobreescritura del metodo mover que tiene la clase `Vehiculo`.

Links:

- 🔗 stefanhenneken.net,iec-61131-3-solid-the-liskov-substitution-principle
- 🔗 hdeleon.net, Principios SOLID: Principio de Sustitución de Liskov LSP
- 🔗 makigas: aprende a programar, SOLID: Principio de Sustitución de Liskov (LSP)
- 🔗 tech.tribalyte.eu, blog-principio-solid-liskov

Link al Video de Youtube 026:

- 🔗 026 - OOP IEC 61131-3 PLC -- SOLID - LSP

December 16, 2023 10:31:59

December 16, 2023 10:31:59

ISP - Principio de Segregación de Interfaz

Principio de Segregación de Interfaz -- (Interface Segregation Principle) ISP :

- El principio de segregación de interfaz establece que una clase no debe implementar interfaces que no utilice y que debe dividirse en interfaces más pequeñas y específicas.
- El principio de segregación de interfaz se debe de mirar desde el lado de los clientes que implementan las interfaces que no tengan metodos y/o propiedades que no tengan sentido para ese cliente.

Ejemplo:

```
INTERFACE I_AveVoladora
// interfaz para las aves voladoras
   METHODS
      Volar : BOOL; // método para volar
END_INTERFACE

INTERFACE I_AveNadadora
// interfaz para las aves nadadoras
   METHODS
      Nadar : BOOL; // método para nadar
END_INTERFACE

INTERFACE I_AveCorredora
// interfaz para las aves corredoras
   METHODS
      Correr : BOOL; // método para correr
END_INTERFACE

FUNCTION_BLOCK Pato IMPLEMENTS I_AveVoladora, I_AveNadadora
// implementa las interfaces I_AveVoladora e I_AveNadadora
VAR_INPUT
   velocidad : REAL;
   alturaMaxima : REAL;
   tiempoBuceo : TIME;
END_VAR

// implementación para el pato
// métodos para volar y nadar

END_FUNCTION_BLOCK

FUNCTION_BLOCK Aguila IMPLEMENTS I_AveVoladora
// implementa la interfaz I_AveVoladora solamente
VAR_INPUT
   velocidad : REAL;
   alturaMaxima : REAL;
END_VAR

// implementación para el águila
// método para volar

END_FUNCTION_BLOCK

FUNCTION_BLOCK Avestruz IMPLEMENTS I_AveCorredora
// implementa la interfaz I_AveCorredora solamente
VAR_INPUT
   velocidad : REAL;
   tiempoCorriendo : TIME;
END_VAR

// implementación para el avestruz
// método para correr

END_FUNCTION_BLOCK
```

- De esta manera, cada clase tiene solo los métodos necesarios y se divide en interfaces más pequeñas y específicas. Además, se utilizan interfaces en lugar de function blocks para implementar el principio de segregación de interfaz.
- Esto permite una mayor flexibilidad y evita la necesidad de implementar métodos innecesarios en una clase.

Links:

- 🔗 stefanhenneken.net,iec-61131-3-solid-the-interface-segregation-principle
- 🔗 IEC 61131-3: SOLID – The Interface Segregation Principle
- 🔗 🤓¡PATRONES de DISEÑO de Typescript que te AYUDARÁN! Guía + Interface Segregation Principle | PT 1.
- 🔗 CodelyTV - Redescubre la programación, Principio de Segregación de Interfaces - SOLID
- 🔗 CodelyTV - Redescubre la programación, Errores comunes al diseñar Interfaces - #SOLID - ISP
- 🔗 hdeleon.net, Principios SOLID: Principio de Segregación de Interfaces ISP
- 🔗 DevExpert - Formación Android & Kotlin, Principio de SEGREGACIÓN de INTERFACES ⚠ Estás usando mal las interfaces [SOLID] 🟡
- 🔗 Segregación de Interfaces - José Luis Rodríguez
- 🔗 makigas: aprende a programar, SOLID: Principio de Segregación de Interfaz (ISP)
- 🔗 tech.tribalyte.eu, blog-principios-solid-interface-segregation

Link al Video de Youtube 027:

- 🔗 027 - OOP IEC 61131-3 PLC -- SOLID - ISP

December 16, 2023 10:31:59

December 16, 2023 10:31:59

DIP - Principio de Inversión de Dependencia

Principio de Inversión de Dependencia -- (Dependency Inversion Principle) DIP :

El principio de inversión de dependencia establece que los módulos de **nivel superior** no deben depender de los módulos de **nivel inferior**, sino que ambos deben depender de abstracciones.

Ejemplo:

```
INTERFACE I_Conexion
// interfaz para la conexión
METHODS
    EstablecerConexion : BOOL; // método para establecer la conexión
END_INTERFACE

FUNCTION_BLOCK ConexionSerial IMPLEMENTS I_Conexion // implementa la
interfaz I_Conexion
// implementación para la conexión serial
END_FUNCTION_BLOCK

FUNCTION_BLOCK ConexionEthernet IMPLEMENTS I_Conexion // implementa la
interfaz I_Conexion
// implementación para la conexión ethernet
END_FUNCTION_BLOCK

FUNCTION_BLOCK Dispositivo

// constructor
Dispositivo(conexion);

// método para establecer la conexión
establecerConexion();

END_FUNCTION_BLOCK
```

- Esto permite que se pueda pasar cualquier objeto que implemente la interfaz `I_Conexion`, lo que cumple con el principio de inversión de dependencias.
- Además, se utiliza el método `EstablecerConexion` definido en la interfaz `I_Conexion`, lo que demuestra cómo se puede utilizar una abstracción (la interfaz) para trabajar con diferentes implementaciones concretas de manera uniforme.

Links:

- 🔗 stefanhenneken.net,iec-61131-3-solid-the-dependency-inversion-principle
- 🔗 Twincontrols__Dependency Injection
- 🔗 tech.tribalyte.eu, blog-principios-solid-dependency

Link al Video de Youtube 028:

- 🔗 028 - OOP IEC 61131-3 PLC -- SOLID - DIP

December 16, 2023 10:31:59

December 16, 2023 10:31:59

UML

UML (Unified Modeling Language) o “Lenguaje Unificado de Modelado”:

Descripción general:

La siglas UML significan (Lenguaje de Modelado Unificado) es un lenguaje gráfico para la especificación, diseño y documentación de software orientado a objetos. Proporciona una base universalmente comprensible para la discusión entre la programación y otros departamentos dentro del desarrollo del sistema.

El lenguaje de modelado unificado en sí mismo define 14 tipos de diagramas diferentes de dos categorías principales:

Diagramas de Estructura:

Los diagramas de estructura representan esquemáticamente la arquitectura del software y se utilizan principalmente para modelado y análisis (por ejemplo, diseño de proyectos, especificación de requisitos del sistema y documentación).

En Codesys y en TwinCAT tendremos el diagrama:

- Diagrama de clase UML (tipo: diagrama de estructura)

Diagramas de Comportamiento:

Los diagramas de comportamiento son modelos ejecutables con sintaxis y semántica únicas a partir de los cuales se puede generar directamente el código de la aplicación (arquitectura basada en modelos).

En Codesys y en TwinCAT tendremos el diagrama:

- UML Statechart (tipo: diagrama de comportamiento)

OOP y UML:

¿La programación orientada a objetos (OOP) y UML siempre tienen que usarse juntos?:

- El uso combinado de OOP y UML ofrece muchos beneficios, aunque no es obligatorio usarlo. La programación de aplicaciones orientada a objetos también es posible sin utilizar UML. Asimismo, UML también se puede utilizar en proyectos de PLC, que no se basan en la programación orientada a objetos (UML Statechart).

¿Cuáles son los beneficios de usar POO y UML juntos?:

- Para aprovechar al máximo la programación orientada a objetos, la estructura de un software orientado a objetos debe diseñarse y crearse antes de la implementación (por ejemplo, qué clases están disponibles, cuál es su relación, qué funcionalidades ofrecen, etc.). Antes, durante y después de la programación, la documentación ayuda a comprender, analizar y mantener el software.
- Como herramienta de análisis, diseño y documentación de software, UML ofrece opciones para planificar, crear y documentar la aplicación. UML es particularmente adecuado para implementaciones orientadas a objetos, ya que el software modular es particularmente adecuado para la representación con la ayuda de un lenguaje gráfico.
- Para la muestra, el diagrama de clases se utiliza para analizar y crear la estructura del programa. Cuanto más modular sea la estructura del software, más fácil y eficiente se puede utilizar el diagrama de clases (por ejemplo, representación gráfica de bloques de funciones separados con métodos individuales para proporcionar las funcionalidades, etc.).
- El diagrama de estado se puede utilizar para especificar la secuencia de un sistema con eventos discretos. Cuanto más coherente sea la orientación a objetos y eventos de la estructura del software, más transparentes y eficaces se podrán diseñar las máquinas de estado (por ejemplo, el comportamiento de los módulos/sistemas se basa en un modelo de estado con estados (como inicio, producción, pausa); dentro de los estados se llaman las funcionalidades correspondientes, las cuales se encapsulan en métodos (como inicio, ejecución, pausa, etc.).

Links UML:

- 🔗 UML 📐 desde -10
- 🔗 Curso UML
- 🔗 Tutorial UML en español
- 🔗 www.plccoder.com, twincat-uml-class-diagram
- 🔗 content.helpme-codesys.com, uml_general
- 🔗 content.helpme-codesys.com, Class Diagram Elements
- 🔗 content.helpme-codesys.com, uml_class_diagram_clarification_terms
- 🔗 online.visual-paradigm.com
- 🔗 www.eclipse.org/papyrus
- 🔗 sourcemaking.com/uml
- 🔗 UML y Enterprise Architect desde cero
- 🔗 Tutorial sobre UML y herramientas CASE

Link al Video de Youtube_29:

- 🔗 029 - OOP IEC 61131-3 PLC -- UML - Descripción

🕓 December 16, 2023 10:31:59

🕓 December 16, 2023 10:31:59

Class UML

Diagrama de Clases en UML:

¿Qué son?:

- El diagrama de clases es un diagrama que muestra **"cómo"** se organiza y estructura un sistema.
- Se enfoca en la estructura interna del sistema y muestra cómo está organizado en términos de clases y objetos.
- Describe cómo se implementará el sistema desde una perpectiva orientada a objetos.

Esta imagen es un Ejemplo en Codesys de Diagrama de Clases:

¿Para qué sirve?:

- Representar la estructura del sistema
- Visualizar relaciones
- Facilitar el diseño del sistema
- Documentar el sistema
- Promover la reutilización

- Facilitar la comunicación
- Base para otros diagramas UML

Representación gráfica:

- Para representar Objetos y Clases, se utiliza el "clasificador".
- 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 atributos/propiedades.
- La sección inferior enumera todos sus operaciones/métodos.

Los Modificadores de acceso de los metodos y las propiedades tienen una Visibilidad y su simbologia es la siguiente:

Symbol	Access modifier
~	INTERNAL
-	PRIVATE
#	PROTECTED
+	PUBLIC

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 a través del diagrama UML.

Los Modificadores de acceso de los metodos y las propiedades se veran segun la simbologia:(Disponible a partir de la versión de TwinCAT 3.1.4026)

TwinCAT 3 PLC | UML Editor – Class diagram **BECKHOFF**

- Access modifier of methods and properties are displayed

Symbol	Access modifier
~	INTERNAL
-	PRIVATE
#	PROTECTED
+	PUBLIC

¿Qué es la asociación?:

- La Asociación entre clases se utiliza para representar relaciones entre diferentes clases. Esta relación significa que una clase conoce a otra y tiene una referencia a ella.

¿Cómo se representa?:

- La asociación se representa con una línea que conecta dos clases en un diagrama de clases.
- Se suele leer como "tiene..."
- Tipo: Adicionalmente se le puede indicar un verbo para especificar el tipo de asociación.
- Direccionalidad: Paréntesis angulares se colocan junto al verbo para indicar direccionalidad
- Multiplicidad: Adicionalmente se pueden utilizar números o rangos que indican cuántas instancias de una clase están asociadas con una instancia de la otra clase.

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
- 🔗 Tutorial - Diagrama de Clases UML
- 🔗 Curso UML. Diagrama de clases I. Vídeo 9
- 🔗 Curso UML. Diagrama de clases II. Vídeo 10
- 🔗 Curso UML. Diagramas de clases III. Traslado a código I. Vídeo 11

Link al Video de Youtube_30:

- 🔗 030 - OOP IEC 61131-3 PLC -- UML - Diagrama de Clases

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Relaciones

En la programación orientada a objetos, un objeto se relaciona con otro para utilizar la funcionalidad y el servicio proporcionados por ese objeto. Esta relación entre dos objetos se conoce como asociación en el diseño de software general orientado a objetos y se representa con una flecha en el Lenguaje Unificado de Modelado o UML.

Dependencias entre Clase:

- Clases: Establecen relaciones para solucionar el problema planteado.
- Existen diferentes tipos de relaciones entre clases.

UML Relaciones Notaciones:

Aquí hay notaciones UML para un tipo diferente de dependencia entre las dos clases.

Clasificación según la Dependencia entre Clases:

- Agregación:
- Composición:

¿Qué es una relación de Asociación?

- Una relación de asociación entre dos clases es una relación **"tiene un"**.
- Ejemplos:
  - Un libro tiene Paginas.
  - Un Humano tiene corazón y sangre.(Composición)
    - Por ejemplo, la clase Humano es una composición de varias partes del cuerpo, incluidas la mano , la pierna y el corazón y la sangre . Cuando

un objeto humano muere, todas las partes de su cuerpo dejan de existir de manera significativa, este es un ejemplo de Composición.

- Un coche tiene un motor y estructura (Agregación ó Composición)
  - Un ejemplo de agregación podria ser el automóvil y sus partes, por ejemplo, motores, ruedas, etc. Las partes individuales del automóvil pueden existir cuando el automovil deja de funcionar. Este ejemplo se podria confundir visto desde otra perpectiva podria ser composición el automóvil y sus partes. Las partes individuales del automóvil no pueden funcionar cuando se destruye..

Otro ejemplo, incluir objetos que pueden existir sin ser parte del objeto principal como un Jugador que es parte de un Equipo, puede existir sin un equipo y también puede formar parte de otros equipos.

Dependencia entre Clases por Agregación:

- Relación débil entre clases.
- A un todo se le agregran componentes.
- El todo puede existir aunque no existan sus componentes.
- Los componentes pueden pertenecer a otros todos.
- Representación en UML:
  - Se representa por un rombo blanco ó sin relleno.

Dependencia entre Clases por Composición:

- Relación estrecha entre clases.
- Componentes forman un todo.
- Todo formado por sus componentes. Si todo se elemina, también se eliminan sus componentes.
- La dependencia entre clases por composición se consigue añadiendo dentro de una clase otra clase en la zona de declaracion de variables (**VAR END_VAR**)

- Representación en UML:
  - Se representa por un rombo negro ó con relleno.
- La composición es una relación como la agregación, pero más fuerte, es decir, un objeto no puede ser ese objeto sin otros objetos, por ejemplo: una silla no puede ser silla sin sus patas, un automóvil no puede ser automóvil sin sus ruedas o su motor, básicamente todos dependen de entre sí.

Diferencias entre Asociación, Agregación y composición:

Tanto la Composición como la Agregación son formas de asociación entre dos objetos, pero existe una diferencia sutil entre composición y agregación , que también se refleja en su notación UML. Nos referimos a la asociación entre dos objetos como Composición cuando una clase posee otras clases y otras clases no pueden existir de manera significativa, cuando el propietario es destruido.

Los programadores a menudo confunden entre Asociación, Composición y Agregación en las discusiones sobre diseño orientado a objetos; esta confusión también hace que la diferencia entre Asociación, Composición y Agregación sea una de las preguntas populares..., solo después de la diferencia entre clase abstracta e interfaz.

La composición es más fuerte que la agregación. En resumen, una relación entre dos objetos se denomina asociación, y una asociación se conoce como composición cuando un objeto posee a otro, mientras que una asociación se conoce como agregación cuando un objeto usa otro objeto.

los tres denotan una relación entre objetos y solo difieren en su fuerza, también puedes verlos como se muestra a continuación, donde la composición representa la forma más fuerte de relación y la asociación es la forma más general.

Niveles de dependencia entre clases:

- 1 (relación uno y solo uno)
- 1..1 (relación uno a uno)
- 0..1 (relación de cero a uno)
- M..N (relación de M hasta N)
- M,N (relación M ó N)
- *(relación de cero a muchos)
- 1..* (relación de uno a muchos y al menos uno)

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**.

Links UML Relaciones:

- 🔗 Relación agregación y composición en Java
- 🔗 Mensajes, agregación y clases abstractas en OOPS.
- 🔗 Diferencia entre Asociación, Composición y Agregación en Java, UML y Programación Orientada a Objetos
- 🔗 Los 5 mejores libros y cursos de UML para que programadores de Java aprendan diseño de software en 2023

Link al Video de Youtube_31:

- 🔗 031 - OOP IEC 61131-3 PLC -- UML - Diagrama de Clases - Relaciones

December 16, 2023 10:31:59

December 16, 2023 10:31:59

StateChart UML

UML State Chart:

- Un diagrama de estado es una máquina que cambia de un estado a otro en tiempo de ejecución.
- Los estados están unidos por transiciones, cada una de las cuales tiene una condición de guardia. Se pueden llamar acciones o métodos tanto en estados como en transiciones. Cuando una condición de guardia obtiene el valor TRUE(evento), se activará la transición. Esto ejecuta las acciones o métodos que pertenecen a la transición y luego cambia al siguiente estado. Las condiciones de guardia son simplemente variables booleanas que reflejan eventos o son una expresión. Los eventos son entradas del usuario de una visualización/interfaz de usuario, E/S, eventos de tiempo o eventos del sistema. Otro evento que a menudo se requiere es el evento de finalización que ocurre cuando se completan las acciones o métodos de un estado.
- Inserta todos los estados requeridos en el editor de diagrama de estado e implementa el control de flujo. Para hacer esto, codifique las condiciones de protección para las transiciones especificando una variable booleana o una expresión ST. Implementas la funcionalidad real del diagrama de estado en las acciones y métodos que se llaman en los estados o durante las transiciones.

Por tanto, los métodos y acciones asignados a un diagrama de estado contienen los algoritmos. Así es como se implementa el concepto de clase orientada a objetos de forma convencional.

Durante la fase de diseño del software, ya puede utilizar el editor de gráficos de estado como herramienta de diseño. Por ejemplo, puede crear un archivo de gráficos (BMP) a partir de un diagrama de estado para agregarlo a una especificación o un documento de diseño.

- Identifica todos los estados que tendrá la máquina.
- Identificar las posibles transiciones de estado de un estado a otro.
- Identificar los eventos que ocurren durante el tiempo de ejecución de la máquina y que desencadenan una transición de estado. Agrupa los eventos relevantes cronológicamente.
- Identifique las ENTRY acciones o métodos DO de, y EXIT para llamar durante un estado.
- Identifique acciones o métodos para llamar durante las transiciones.

- Definir el comportamiento en caso de error.

Diagrama de Estado:

- Un diagrama de estado es un formalismo gráfico con el que se puede programar gráficamente una máquina de estados finitos. Una máquina de estados es un sistema que se encuentra continuamente en uno de un número finito de estados en tiempo de ejecución. Las acciones se pueden ejecutar en cada estado. Cuando ocurre un evento, se produce una transición al siguiente estado. También se pueden realizar acciones durante la transición.
- En CODESYS, TwinCAT un diagrama de estado es una POU que se crea en el lenguaje de implementación de Statechart. Este tipo de POU se identifica por el_uml_icon_statechart.png símbolo en la vista POU o en la vista Dispositivos. Puede crear programas, bloques de funciones, funciones, métodos, acciones o propiedades como diagramas de estado. El editor proporciona elementos para estados, pseudoestados y transiciones.
- Tanto los estados como las transiciones pueden llamar métodos o acciones. Los pseudoestados son elementos de control que se utilizan para controlar el proceso, pero no invocan ninguna acción ni método.

Imagen UML TwinCAT StateChart Simbols:

	Estado de inicio
	Estado final
	Estado
	Estado compuesto
	Bifurcar/Unir
	Elección
	Transición
	Transición de finalización
	Transición de excepción
	Nota

Imagen UML Codesys StateChart Simbols:

Estados:

- : Estado
- : Estado rápido
- : Estado compuesto

Pseudo estados:

- : Estado inicial
- : Estado final
- : Bifurcar/Unir
- : Elección

Transiciones:

- : Transición
- : Transición de finalización
- : transición de excepción

Imagen UML Codesys StateChart CoffeMachine:

Imagen UML TwinCAT StateChart:

- En TwinCAT para utilizar UML State Chart, se podra utilizar con el TF1910, el cual tendra un coste de licencia según el Performance Level (PL)
- En Codesys para utilizar UML State Chart, se podra utilizar con el ide Codesys Professional Developer Edition, según el informe de caracteristicas realizado en otoño de 2023 (pongo link mas abajo) UML State Chart sera discontinuado en el Q1 2024 (minuto del video 35:09), desde mi punto de vista es un gran error ya que no se deben de perder las diferentes posibilidades de programación si no todo lo contrario se deben de mantener y añadir nuevos

lenguajes y formas de programar para que sea el usuario el que decida que utilizar...

Links UML State Chart:

- 🔗 infosys.beckhoff.com, tf1910_tc3_uml
- 🔗 content.helpme-codesys.com, UML State Chart
- 🔗 content.helpme-codesys.com, uml_sc_simple_machine
- 🔗 content.helpme-codesys.com, _ex_uml_sc_coffee_machine
- 🔗 www.infoplc.net, el-nuevo-uml-state-chart-en
- 🔗 CODESYS Feature Briefing - Fall 2023

Link al Video de Youtube_32:

- 🔗 032 - OOP IEC 61131-3 PLC -- UML - Diagrama de Estado - StateChart

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Tipos de Diseño para programacion OOP

Análisis y Diseño de un sistema orientado a objetos:

- Durante el desarrollo de un sistema se recomienda seguir la siguiente fases:

Proceso de Desarrollo:

- Análisis:
  - Vamos a pensar en el problema y vamos a identificar los elementos con los cuales vamos a trabajar a lo largo del desarrollo. Una vez que hemos identificado estos elementos de manera superficial pasamos a la siguiente fase.
- Diseño:
  - Durante esta fase vamos ya a definir cual va a ser el comportamiento de cada uno de los elementos que habiamos definido en la fase de analisis, tambien estableceremos las relaciones entre elementos.
  - Para esta fase de diseño podemos utilizar los Diagramas UML, mediante estos diagramas vamos a poder modelar lo que va a ser finalmente el sistema. Esta fase se puede optar por no hacerla pero si se esta diseñando un sistema mediano a gran tamaño y/ó con cierta complejidad se recomienda no saltarse este paso.
  - Los Diagrams UML mas utilizados son:
    - Diagrama de Clases:
      - Representar cada una de las clases de nuestro sistema y sus relaciones entre ellas, tambien representaremos dentro de cada clase los atributos, propiedades y metodos que va a contener cada clase.
    - Diagrama de Casos de uso:
      - Sirve basicamente para modelar el comportamiento de un sistema, subsistema o una clase. Este sistema también va a incluir a los usuarios y su interacción con las funciones del sistema.
    - Diagrama de Secuencia:
      - El diagrama de Secuencia es un tipo de diagrama usado para modelar interacción entre objetos en un sistema según UML.

    - Los diagramas de Secuencia son una solución de modelado dinámico popular en UML porque se centran específicamente en líneas de vida o en los procesos y objetos que coexisten simultáneamente, y los mensajes intercambiados entre ellos para ejecutar una función antes de que la línea de vida termine.

- Codificación:
    - En esta fase escribiremos el codigo que va a dar vida a lo que hemos analizado y diseñado previamente.

🚀 ¿ Qué es una Unidad de Software ?

En el desarrollo de software, comprender qué es una **"Unidad de Software"** es esencial, pero no siempre es tan sencillo como parece. ¿Cuáles son los desafíos y las soluciones para definir este componente crítico?

🧩 Definición:

Una unidad de software es el nivel atómico de un componente de software, elegible para pruebas independientes. Encapsula funcionalidades específicas y está diseñado para desarrollo y pruebas independientes. Varias unidades de software forman colectivamente un componente de software.

🎯 Caracteristicas de la Unidad de Software:

1️⃣ **Indepencia:** Puede diseñarse, implementarse y probarse por separado, lo que lo hace fácil de usar para los desarrolladores.

2️⃣ **Enfoque de Requisitos:** Cumple de forma exhaustiva un conjunto de requisitos bien definidos, asegurando la trazabilidad.

3️⃣ **Auto-Cohesion:** Es altamente atómico y se centra en una única funcionalidad con dependencias mínimas.

4️⃣ **Reutilización:** Está diseñado para ser reutilizado si es necesario en múltiples aplicaciones de software para una mayor eficiencia.

📐 Tamaño y Complejidad:

Aquí está el principal desafío: definir el tamaño ideal y gestionar la complejidad. No existe una ciencia exacta, pero se pueden proporcionar pautas que incluyen:

1️⃣ **Analizabilidad:** Garantizar que el tamaño de la unidad permita un fácil análisis y comprensión de los requisitos asignados.

2️⃣ **Consideración de funcionalidad:** Elegir el algoritmo correcto teniendo en cuenta las desventajas limitaciones del software integrado (por ejemplo, recursos de hardware disponibles)

3️⃣ **Acoplamiento Inteligente:** Agrupar funcionalidades altamente dependientes dentro de una unidad mientras se separan funciones desacopladas en diferentes unidades.

4️⃣ **Criterios Técnico:** Uso de métricas, herramientas de análisis de complejidad, límites de recursos y técnicas de partición para guiar el diseño de la unidad de software.

En el panorama del software en constante evolución, comprender los desafíos que plantea la definición de unidades de software es fundamental para crear sistemas de software robustos, eficientes y escalables.

Ingenieria de desarrollo para la programación OOP:

- Diseño por Unidad,Componente,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):

- I_InputDigital(p_On, p_Off)
- I_OutputDigital(M_ON, M_OFF)
- I_InputAnalog
- I_OutputAnalog
- I_Run:(M_Start, M_Stop)

- FBTimer
- FCAnalogSensor
- FBGenericUnit
- Objects Composition (Composicion de Objetos).
- Conceptos básicos del lenguaje de programación de texto estructurado (ST).
- UDT (estructuras).
- Diseño Modular.
- Polimorfismo.
- Patrones de Diseño: Patron de estado Avanzado.
- Envoltorios y caracteristicas.
- Diseño en Capas.
- Objetivo Proyecto final que cubre un problema del mundo real a resolver utilizando programación orientada a objetos OOP

Links Tipos de Diseño para programación de PLC:

- 🔗 www.btc-embedded.com, component-based-architecture-in-simulink-for-efficient-unit-and-integration-test

Link al Video de Youtube_33:

- 🔗 033 - OOP IEC 61131-3 PLC -- Tipos de Diseño para programación OOP

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patrones de Diseño

El proyecto con todos los patrones de Diseño lo podremos encontrar en:

📚👉 Proyecto TwinCAT Patrones de Diseño

📚👉 Ficheros PLCOpen XML Patrones de Diseño

Imagen de los 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.

```
“Los patrones de diseño son
descripciones de objetos y clases
conectadas que se personalizan para
resolver un problema de diseño
general en un contexto particular”.
- Gang of Four
```

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.).

Mediador

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.

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.

Copyright © 2007 Jason S. McDonald
http://McDonaldLand.wordpress.com

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

15 Design Patterns in Simple Words

Adapter

Convert the interface of a class to another interface that clients expect.

interface, class, conversion

Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

implementation, abstraction, decoupling, independent variation

Observer

Notify multiple objects when the state of another object changes.

multiple objects, state change, notification

Decorator

Add behavior to an object dynamically without affecting the behavior of other objects.

object behavior, dynamic addition, no effect on other objects

Builder

Simplify the creation of complex objects with many optional parameters.

complex objects, optional parameters, simplified creation

Factory

Dynamically create objects based on input, configuration, or business rules.

object creation, dynamic, input, configuration, business rules

Template Method

Define the skeleton of an algorithm in a base class and let subclasses override specific steps.

algorithm, skeleton, base class, specific steps, subclass

Command

Encapsulate a request as an object and pass it to invokers to execute.

encapsulation, request, object, invokers, execution

Proxy

Control access to an object by creating a surrogate that handles the request.

access control, surrogate, request handling

Iterator

Traverse elements of a collection without exposing its underlying implementation.

collection, element traversal, implementation hiding

Facade

Provide a unified interface to a set of interfaces in a subsystem to simplify their usage.

simplified usage, subsystem, interface, unification

Composite

Compose objects into a tree structure to represent part-whole hierarchies.

tree structure, part-whole hierarchy, object composition

Flyweight

Share objects to support large numbers of fine-grained objects efficiently.

object sharing, fine-grained objects, efficiency

Visitor

Separate the algorithm from the object structure it operates on.

algorithm separation, object structure, operation

State

Alter an object's behavior when its state changes.

State change, notification

Keivan Damirchi

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:

- Los patrones de Creación abstraen la forma en que se crean los objetos, de forma que permite tratar las clases a crear de forma genérica apartando la decisión de qué clases crear o como crearlas. Pero los Patrones de Diseño son conceptos aplicables directamente en la producción de software, cualquier abstracción no se queda en el aire como una entelequia que solo sirve para dar discursos, así: Según a donde quede desplazada dicha decisión se habla de Patrones de Clase (utiliza la herencia para determinar la creación de las instancias, es decir en los constructores de las clases) o Patrones de Objeto (es en métodos de los objetos creados donde se modifica la clase)
- Patrones de Creación de Clase:
  - Factoría Abstracta
  - Builder
- Patrones de Creación de Objeto:
  - Método Factoría
  - Prototipo
  - Singleton
  - Object Pool
- Builder*
- Singleton *
- Dependency Injection
- Service Locator
- Abstract Factory*
- Factory Method *

Patrones Estructurales:

Tratan la relación entre clases, la combinación clases y la formación de estructuras de mayor complejidad.

- Adapter
- Data Access Object (DAO)

- Query Object
- Decorator
- Bridge

Patrones de Comportamiento:

Los patrones de comportamiento hablan de como interaccionan entre si los objetos para conseguir ciertos resultados. Los principales patrones de comportamiento son:

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

Los patrones de diseño son soluciones reutilizables para problemas comunes de diseño de software. Proporcionan una forma para que los desarrolladores de software resuelvan problemas comunes de manera consistente y eficiente, sin tener que reinventar la rueda cada vez.

Beneficios de usar Patrones de Diseño =>

• Reusabilidad: Evite reinventar la rueda cada vez.

• Escalabilidad: Diseño de software flexible y adaptable.

• Capacidad de mantenimiento: Código más fácil de modificar y depurar.

• Estandarización: Vocabulario común y estructura a través de diferentes proyectos.

• Colaboración: más fácil para varios desarrolladores trabajar en el mismo código base.

Algunos patrones de diseño de uso común =>

• **Patrón de estrategia:** el patrón de estrategia se utiliza para definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables.

Por ejemplo, imagina que tienes un juego con diferentes tipos de personajes, cada uno con sus propias habilidades únicas. El patrón de estrategia le permitiría definir un conjunto de estrategias (es decir, algoritmos) para cada tipo de personaje y luego cambiar fácilmente entre ellas según sea necesario.

• **Patrón de observador:** el patrón de observador se utiliza para notificar a los objetos cuando hay un cambio en otro objeto.

Por ejemplo, imagine que tiene una aplicación meteorológica que necesita notificar a sus usuarios cuando cambia la temperatura. El patrón de observador le permitiría definir un conjunto de observadores (es decir, los usuarios) y luego notificarles cuando cambie la temperatura.

• **Patrón de decorador:** el patrón de decorador se utiliza para agregar funcionalidad a un objeto de forma dinámica, sin cambiar su estructura original.

Por ejemplo, imagine que tiene un automóvil y desea agregarle un sistema de navegación GPS. El patrón decorador le permitiría agregar el sistema GPS sin tener que modificar el propio automóvil.

• **Patrón de comando:** el patrón de comando se usa para encapsular una solicitud como un objeto, lo que permite que se almacene, pase y ejecute en un momento posterior.

Por ejemplo, imagina que tienes un sistema de automatización del hogar que te permite controlar las luces, el termostato y otros dispositivos. El patrón de comando le permitiría encapsular cada comando (por ejemplo, encender las luces), almacenarlo como un objeto y ejecutarlo más tarde.

• **Patrón de fábrica:** el patrón de fábrica se utiliza para crear objetos sin exponer la lógica de creación al cliente.

Por ejemplo, imagina que tienes un juego con diferentes niveles, cada uno con su propio conjunto de enemigos. El patrón de fábrica te permitiría crear enemigos para cada nivel sin exponer la lógica de creación al cliente.

• **Patrón compuesto:** el patrón compuesto se utiliza para crear una estructura de objetos en forma de árbol, donde los objetos individuales y los grupos de objetos se tratan de la misma manera.

Por ejemplo, imagine que tiene un sistema de archivos, donde los archivos y los directorios se tratan de la misma manera. El patrón compuesto le permitiría tratar archivos y directorios individuales como el mismo tipo de objeto y crear una estructura similar a un árbol de todo el sistema de archivos.

Además de los Patrones de Diseño tenemos:

- **Patrones de Arquitectura:** Formas de descomponer, conectar y relacionar sistemas, trata conceptos como: niveles, tuberías y filtros. Es un nivel de abstracción mayor que el de los Patrones de Diseño.
- **Patrones de Programación (Idioms Patterns):** Patrones de bajo nivel acerca de un lenguaje de programación concreto, describen como** implementar cuestiones concretas.
- **Patrones de Analisis:** Conjunto de reglas que permiten modelar un sistema de forma satisfactoria.
- **Patrones de Organizacionales:** Describen como organizar grupos humanos, generalmente relacionados con el software.
- **Otros Patrones de Software:** Se puede hablar de patrones de Programación concurrente, de Interfaz Gráfica, de Organización de Código, de Optimización de Código, de Robustez de Código, de Fase de Prueba.

Links de Patrones de Diseño:

- 🔗 refactoring.guru/es/design-patterns
- 🔗 0w8States/PLC-Design-Patterns
- 🔗 github.com/Aliazzzz/Applied-Design-Patterns-in-CODESYS-V3
- 🔗 sourcemaking.com/design_patterns
- 🔗 Coding Together ES, Patrones de Diseño
- 🔗 medium.com, design-patterns
- 🔗 8 Design Patterns EVERY Developer Should Know
- 🔗 ¿Qué son los PATRONES de DISEÑO?
- 🔗 Aprende los principales patrones de diseño con .NET y C#. Temario curso

Link al Video de Youtube_34:

- 🔗 034 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Introducción

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Factory Method

El 👉**patrón de diseño del método de fábrica** es uno de los 👉**patrones de diseño creacional** que se ocupa de los problemas de creación de objetos durante el diseño y desarrollo de software.

Estados de patrón de método Factory/Factory que definen una interfaz o clase abstracta para crear un objeto, pero permiten que las subclases decidan qué clase instanciar/crear. En lugar de crear instancias de clases en la aplicación principal, las subclases son responsables de realizar esta tarea.

El método de fábrica también se conoce como Constructor Virtual.

El diagrama de clases UML del método de fábrica es el siguiente:

Implementación de ejemplo practico en TwinCAT 3:

Links de Patrones de Diseño Método de Fábrica:

- 🔗 refactoring.guru, factory-method
- 🔗 twincontrols.com, Factory Method Design Pattern

Link al Video de Youtube_35:

- 🔗 Factoy Method Design Pattern Youtube

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Abstract Factory

- Abstract Factory es un patrón de diseño creacional que le permite producir familias de objetos relacionados sin especificar sus clases concretas.
- En el 👉**patrón de fábrica abstracto** , una superfábrica que genera más fábricas es el centro de los patrones de fábrica abstractos. Otro nombre para esta instalación es fábrica de fábricas. Este tipo de patrón de diseño entra en la categoría de patrón de creación porque ofrece una de las mejores formas de crear un objeto.
- Una fábrica de objetos relacionados se crea a través de una interfaz utilizando el patrón de fábrica abstracto sin necesidad de declarar explícitamente las clases de los objetos. Según el patrón Factory, cada fábrica producida puede entregar objetos.

El diagrama de clases UML de la fábrica abstracta es el siguiente:

Implementación de ejemplo practico en TwinCAT 3:

Links de Patrones de Diseño Fábrica Abstracta:

- 🔗 iec-61131-6-abstract-factory-english,stefanhenneken.net
- 🔗 Abstract Factory Design Pattern
- 🔗 refactoring.guru,abstract-factory

Link al Video de Youtube_36:

- 🔗 Factoy Abstract Design Pattern Youtube

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Builder

- 👉**Builder** es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso.
- El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

El diagrama de clases UML del Patrón Creacional Builder es el siguiente:

Implementación 1 del Patrón de Diseño Creacional Builder, ejemplo practico en TwinCAT 3:

Implementación 2 de Patrón de Diseño Creacional Builder, ejemplo practico en TwinCAT 3:

Links de Patrones de Diseño Builder:

- 🔗 github.com,Aliazzzz,Applied-Design-Patterns-in-CODESYS-V3
- 🔗 0w8States/PLC-Design-Patterns
- 🔗 refactoring.guru, builder

Link al Video de Youtube_37:

- 🔗 Builder Design Pattern Youtube

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Prototype

- 👉**Prototipo** es un patrón de diseño creacional que le permite copiar objetos existentes sin que su código dependa de sus clases, si los atributos son privados no se podran copiar tal cual.
- Duplicamos objetos a partir de unos que ya tenemos.
- El proceso de clonación se delega a los propios objetos objetos prototipos que estan siendo clonados.
- Para ello se realiza a partir de una interface o clase abstracta, las clases prototipo implementan la inteface, por lo tanto estan obligadas a implementar los metodos y propiedades de la interface, en la interface añadimos un metodo llamado clonar que sera el encargado de devolver el propio objeto, con esto conseguimos no tener la neceseidad de acoplar la logica de clonación de las clases lo tendremos desacoplado.
- Este patrón es util cuando crear objetos sea muy costoso en terminos de tiempo y de recursos, o que los objetos tienen una estructura compleja que sea dificil de replicar con constructores, también cuando queremos un objeto configurado de una manera especifica con configuracions similares.
- Utilice el patrón de prototipo si desea crear un objeto con los mismos valores de propiedad de otro objeto existente.

El diagrama de clases UML del Patrón Creacional Prototipo es el siguiente:

Implementación del patrón de Diseño Creacional Prototipo, ejemplo practico en TwinCAT 3:

Links de Patrones de Diseño Prototipo:

- 🔗 softwareparticles.com, design-patterns-prototype
- 🔗 0w8States/PLC-Design-Patterns
- 🔗 refactoring.guru, prototype
- 🔗 Aprende los principales patrones de diseño con .NET y C#. Patron Prototype

Link al Video de Youtube_38:

- 🔗 Prototype Design Pattern Youtube

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Singleton

- 👉**Singleton** es un patrón de diseño creacional que le permite garantizar que una clase tenga solo una instancia, al tiempo que proporciona un punto de acceso global a esta instancia.

El diagrama de clases UML del Patrón Creacional Singleton es el siguiente:

- La clase Singleton declara el método estático getInstance que devuelve la misma instancia de su propia clase.
- El constructor de Singleton debe estar oculto en el código del cliente. Llamar al getInstance método debería ser la única forma de obtener el objeto Singleton.

Implementación de Singleton ejemplo practico en TwinCAT 3:

Links de Patrones de Diseño Singleton:

- 🔗 github.com,Aliazzzz,Applied-Design-Patterns-in-CODESYS-V3
- 🔗 refactoring.guru, singleton

Link al Video de Youtube_39:

- 🔗 Singleton Design Pattern Youtube

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Adapter

- 🔗 Patrón de diseño Adapter en C#

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Bridge

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Composite

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Decorator

Links de Patrones de Diseño Decorador:

- 🔗 Decorator Design Pattern
- 🔗 github.com,Aliazzzz,Applied-Design-Patterns-in-CODESYS-V3
- 🔗 🤩PATRONES de DISEÑO con Typescript - Decorator Pattern | PT 2.

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Facade

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Flyweight

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Proxy

- 🔗 github.com,Aliazzzz,Applied-Design-Patterns-in-CODESYS-V3

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Chain of Responsibility

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Command

- 🔗 github.com,Aliazzzz,Applied-Design-Patterns-in-CODESYS-V3

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Iterator

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Mediator

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Memento

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Observer

Links de Patrones de Diseño:

- 🔗 Observer Design Pattern
- 🔗 github.com,Aliazzzz,Applied-Design-Patterns-in-CODESYS-V3

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron State

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Strategy

Links de Patrones de Diseño:

- 🔗 TwinCAT with Head First Design Patterns Ch.1 - Intro/Strategy Pattern
- 🔗 PATRONES de DISEÑO en PROGRAMACIÓN FUNCIONAL

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Template Method

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Patron Visitor

Links de Patrones de Diseño:

- 🔗Design Patterns - The Visitor Pattern

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Librerias

Librerias:

Cuando desarrollas un proyecto, ¿qué haces cuando quieres reutilizar el mismo programa para otro proyecto? Probablemente el más común es copiar y pegar. Esto está bien para proyectos pequeños, pero a medida que crece la aplicación, las bibliotecas nos permiten administrar las funciones y los bloques de funciones que hemos creado.

Mediante el uso de bibliotecas, podemos administrar el software que hemos creado en múltiples proyectos. En primer lugar, es un hecho que diferentes dispositivos tendrán diferentes funciones, pero aun así, siempre habrá partes comunes. En el mundo del desarrollo de software, ese concepto de gestión de bibliotecas es bastante común.

¿Cuáles son las ventajas de usar la biblioteca?

- El software es modular, por ejemplo, si tengo software para cilindros, puedo usar la biblioteca de cilindros, y si tengo software para registro, puedo usar la biblioteca de registro.
- Cada biblioteca se prueba de forma independiente.

Links Librerias:

- 🔗 soup01.com,beckhofftwincat3-library-management
- 🔗 PLC programming using TwinCAT 3 - Libraries (Part 11/18)
- 🔗 help.codesys.com,_cds_obj_library_manager/
- 🔗 help.codesys.com,_cds_library_development_information/
- 🔗 help.codesys.com,_tm_test_action_libraries_addlibrary
- 🔗 CODESYS Webinar Library Management Basics
- 🔗 CoDeSys - How to add libraries and more with Machine Control Studio.
- 🔗 SPT Application Framework Documentation
- 🔗 Beckhoff.SPT Framework Tutorial Part1.EN
- 🔗 Tutorial #17: Eigene CoDeSys Bibliotheken erstellen
- 🔗 CoDeSys Library Management & Creation

- 🔗 shaswatraj.com, codesys-library-management-step-by-step-guide

Link al Video de Youtube xxx:

```
!!! mirar este video por ejemplo para empezar como concepto a realizar las librerias!!!
```

- 🔗 Nguyễn Vỹ, Beckhoff & PLC - OOP Properties| TWINCAT3

- 🔗 The Benefits of OOP in PLC Programming: A Case Study with a Conveyor System

December 16, 2023 10:31:59

December 16, 2023 10:31:59

CI/CD - Integración y entrega continua

Links CI/CD:

- 🔗 www.redhat.com,what-is-ci-cd
- 🔗 ¿QUE ES CI/CD?
- 🔗 Integración y Entrega Continua desde cero | Live 🔴

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Git

Links Git:

- 🔗 Webinar TwinCAT con Git
- 🔗 PLC programming using TwinCAT 3 - Version control (Part 13/18)
- 🔗 soup01.com, beckhofflets-try-source-control-with-git-2
- 🔗 CoDeSys Git PLC Version Control #codesys #git
- https://github.com/github/gitignore/blob/main/TwinCAT3.gitignore

December 16, 2023 10:31:59

December 16, 2023 10:31:59

TDD - Desarrollo guiado por pruebas

Desarrollo Guiado por Pruebas:

La clave del TDD o Test Driven Development es que en este proceso se escriben las pruebas antes de escribir el código. Este sistema consigue no solo mejorar la calidad del software final, sino que, además, ayuda a reducir los costes de mantenimiento.

La premisa detrás del desarrollo dirigido por pruebas, según Kent Beck, es que todo el código debe ser probado y refactorizado continuamente. Kent Beck lo expresa de esta manera: **Sencillamente, el desarrollo dirigido por pruebas tiene como objetivo eliminar el miedo en el desarrollo de aplicaciones**.

- Está creando documentación, especificaciones vivas y nunca obsoletas (es decir, documentación).
- Está (re)diseñando su código para hacerlo y mantenerlo fácilmente comprobable. Y eso lo hace limpio, sin complicaciones y fácil de entender y cambiar.
- Está creando una red de seguridad para hacer cambios con confianza.
- Notificación temprana de errores.
- Diagnóstico sencillo de los errores, ya que las pruebas identifican lo que ha fallado.

El Ciclo y las Etapas del TDD:

El Desarrollo Dirigido por Pruebas significa pasar por tres fases. Una y otra vez.

- Fase roja: escribir una prueba.
- Fase verde: hacer que la prueba pase escribiendo el código que vigila.
- Fase azul: refactorizar.

Niveles de Testing:

Figure 3: Levels of testing (Burnstein, 2003, p. 134)

- El Desarrollo Guiado por Pruebas debe de empezarse a implementar lo mas temprano posible en el desarrollo del Software.
- Las pruebas solo son útiles cuando se ejecutan. Configurar un sistema de integración continua para construir y probar cada componente cada vez que hay un cambio en el código fuente es el camino a seguir.

¿Como Conseguimos un Desarrollo Guiado por Pruebas TDD en PLCs?:

Vamos a ver una lista de todas las implementaciones que he encontrado para conseguirlo:

-

Links Desarrollo Guiado por Pruebas:

- 🔗 www.nimblework.com, desarrollo-dirigido-por-pruebas-tdd
- 🔗 leanpub.com/tdd-ebook/read
- 🔗 Pruebas unitarias y Test-Driven Development | Ejemplo desde cero
- 🔗 programacionymas.com, tipos-de-testing-en-desarrollo-de-software

Link al Video de Youtube_NNN:

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Pruebas Unitarias

Links Pruebas Unitarias:

- 🔗 https://tcunit.org/
- 🔗 https://github.com/tcunit/TcUnit
- 🔗 https://github.com/PeterZerlauth/Testing
- 🔗
- 🔗
- 🔗
- 🔗
- 🔗 C#: Desarrollo Test Driven
- 🔗 Visual Studio: Unit Tests esencial

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Pruebas Integracion

Pruebas de Integración:

Las pruebas de integración son una parte crítica del proceso de desarrollo de software. Ayuda a los desarrolladores a garantizar que los componentes externos de un sistema funcionen como se espera.

¿cuál es la diferencia entre las pruebas de integración y las pruebas unitarias?

Las pruebas de integración juegan un papel fundamental en la evaluación del funcionamiento adecuado de la infraestructura de soporte de una aplicación, incluida la base de datos, el sistema de archivos y las dependencias externas.

A diferencia de las pruebas unitarias que se centran en unidades de código individuales de forma aislada, las pruebas de integración simulan dependencias externas del mundo real para verificar que los componentes funcionen juntos sin problemas y para detectar cualquier defecto que pueda surgir durante la integración.

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Simulacion

- Machine simulation with CODESYS applications EN - CODESYS Technology Day 2023

December 16, 2023 10:31:59

December 16, 2023 10:31:59

Links OOP

Links de OOP:

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

- https://github.com/benhar-dev/twincat-resources

December 16, 2023 10:31:59

December 16, 2023 10:31:59