Requisitos

👨‍🎓 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...

Introdução

📖 Curso Programación Orientada a Objetos Youtube -- OOP :

OOP_Titulo 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)**

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

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

Links:

- 🔗 Codesys admite OOP
- 🔗 Beckhoff TwinCAT 3 admite OOP
- 🔗 Why Object Oriented PLC Programming is Essential for Industrial Automation

Link al Video de Youtube 001:

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

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.

programacion_procedimental

- 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.
```

ventajasprogramacionoop

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 al Video de Youtube 002:

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

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

Tipos de Dados

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

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_
- 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

Tipos de variáveis e variáveis especiais

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.
- 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

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

Modificadores de acesso

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.

Tabela de Modificadores de acesso

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

Texto Estruturado Estendido

ExST - Texto Estructurado Extendido:

- 🔗 Structured Text and Extended Structured Text (ExST), infosys.beckhoff.com

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**)

OOP_Principios

Links de Principios OOP:

- 🔗 github.com/Aliazzzz/OOP-Concept-Examples-in-CODESYS-V3

Abstração

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.

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 Abstracción:

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

Encapsulamento

Encapsulamiento:

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

Links Encapsulacion:

- 🔗 www.plccoder.com,encapsulation

Herança

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. ¿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.

- 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

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. 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, 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)

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)

UML

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

Descripción general:

UML (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)

Links 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

Class UML

Diagrama de Clases en UML:

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

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

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

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

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)

UML_ClassDiagram Access Modifier

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

Relações

.Relaciones:

Vamos a ver 2 tipos de relaciones:

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

StateChart UML

UML State Chart:

Links UML State Chart:

- 🔗 content.helpme-codesys.com, UML State Chart

Tipos de Design para programação de PLC

Tipos de Diseño para programacion de PLC:

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

Units: (Ejemplo de Unidades): - 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

!!! puntos que se pueden incluir en el curso!!!: - Objects composition (Composicion de Objetos)

- Basic of Structured Text programming Language
- UDT (estructuras)
- Modular Design
- Polymorphism
- Advanced State Pattern
- Wrappers and Features
- Layered Design
- Final Project covering a real-world problem to be solved using OOP
- Texto estructurado (ST), Texto estructurado extendido (ExST)

Padrões de Design

PATRONES DE DISEÑO:

Designpatterns

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

Patrones_de_Diseño_Estructural

Patrones_de_Diseño_de_Comportamiento

Design_patterns

Design_patterns_15

Design_Patterns_Use_Cases_6

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

Padrão Método de fábrica

Links de Patrones de Diseño:

- 🔗 Factory Method Design Pattern

Padrão Fábrica Abstrata

Links de Patrones de Diseño:

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

Padrão Decorador

Links de Patrones de Diseño:

- 🔗 Decorator Design Pattern

Padrão de Estratégia

Links de Patrones de Diseño:

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

Padrão Observador

Links de Patrones de Diseño:

- 🔗 Observer Design Pattern

Padrão Visitante

Links de Patrones de Diseño:

- 🔗Design Patterns - The Visitor Pattern

Livrarias

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.

Links OOP

Links de OOP:

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

TDD - Desenvolvimento Orientado a Testes

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:

Levels of Testing

The_Pyramid_Of_Test

- 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

Testes de Unidade