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:

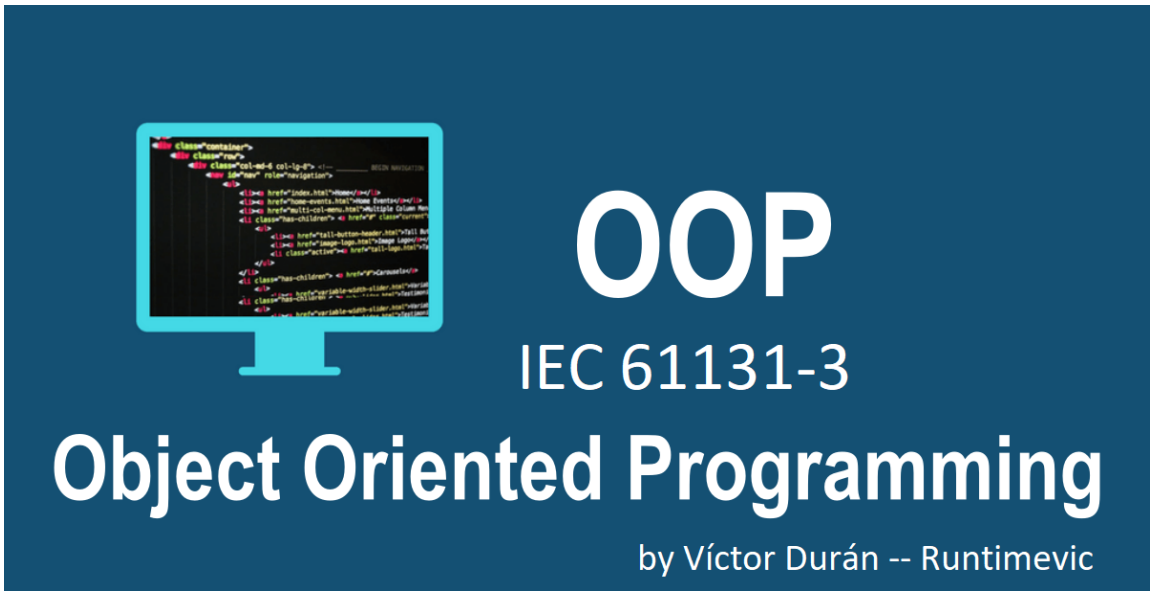
- Clonar el repositorio de 🔗GitHub:

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

  ó utilizar por ejemplo GitHub Desktop para Clonar el repositorio de GitHub...

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.

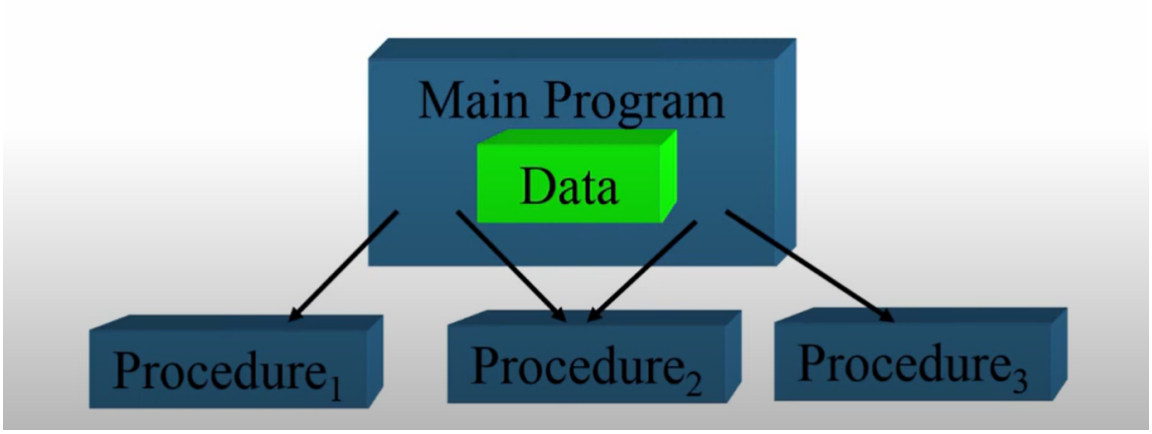
¿Cómo pensar en Objetos?

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

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.
```

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.

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.

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.

May 12, 2024 19:25:39

May 12, 2024 19:25:39

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

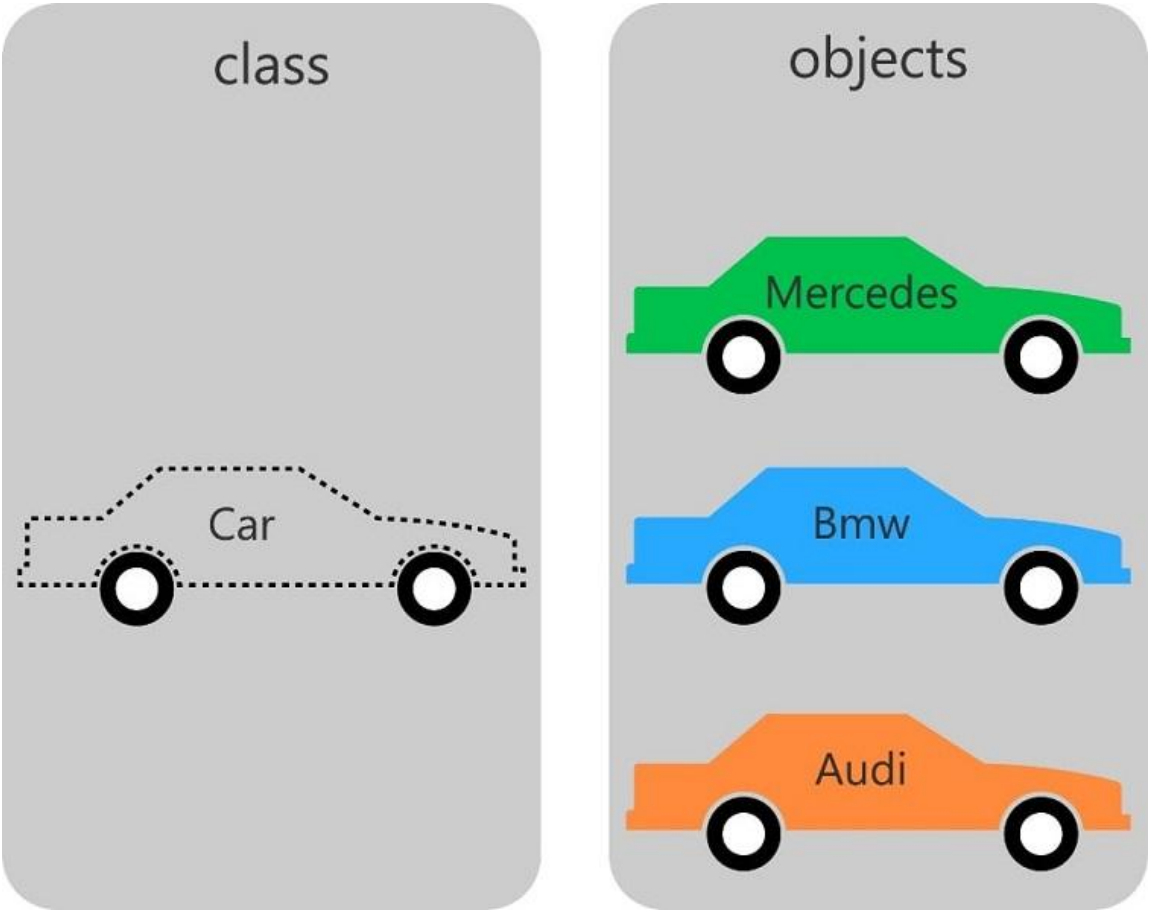
May 12, 2024 19:25:39

May 12, 2024 19:25:39

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...
```

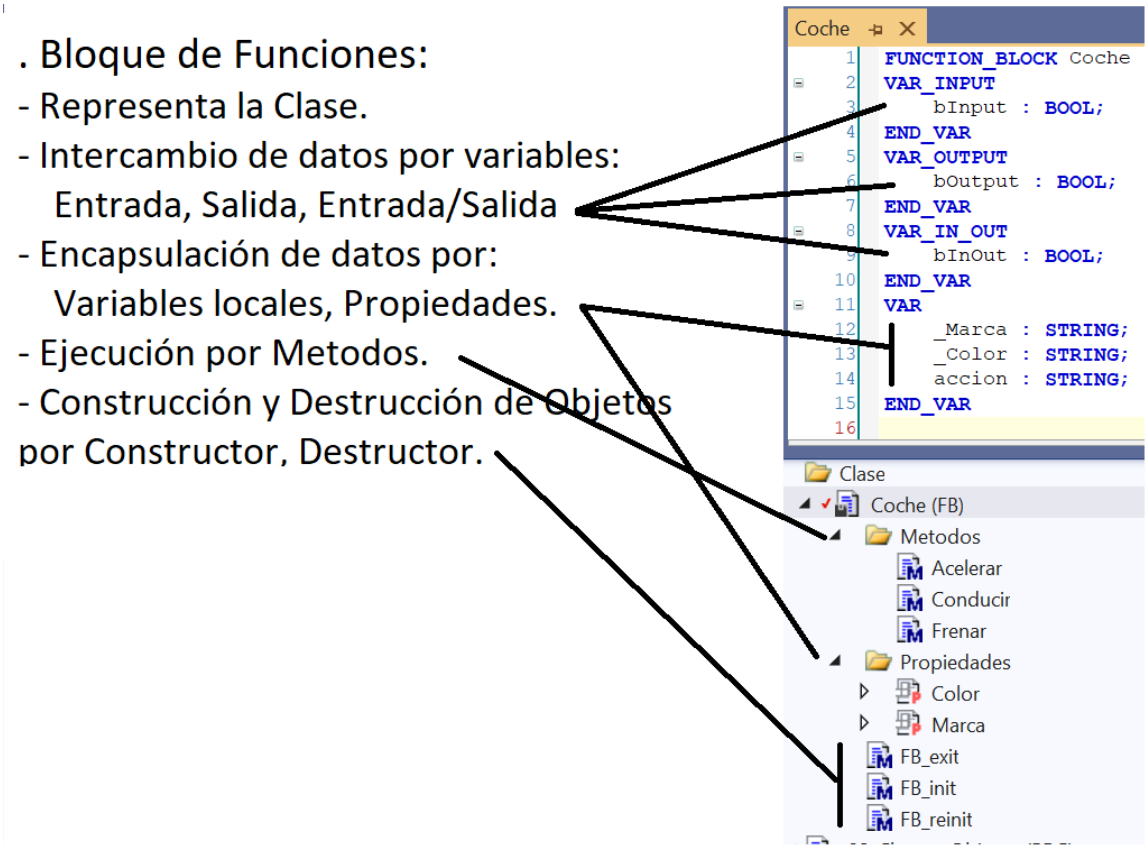
Representacion de la Clase Coche en STL OOP IEC 61131-3

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.

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

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**:

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

Metodo

METHOD:

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

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

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

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

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

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

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

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

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

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.

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**:

Metodo tipos de variables de retorno

Tipos de variables de retorno:

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

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

Retorno por STRUCT:

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

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

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

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

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

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

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

Ejemplo:

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

Objeto Propiedad

Propiedades:

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

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

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

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

Propiedades: Getters & Setters:

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

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

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

En el Objeto Propiedad es obligatorio que retorne un valor.

Herencia Bloque de Funcion

Herencia Bloque de Funcion:

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

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

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

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

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

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

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

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
```

Herencia Interface

Herencia Interface:

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

Cree una interface que amplíe otra interface mediante la extensión:

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

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

```
1   INTERFACE I_Sub2 EXTENDS I_Sub1
```

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

Links:

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

Link al Video de Youtube 008:

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

May 12, 2024 19:25:39

May 12, 2024 19:25:39

THIS puntero

THIS^ puntero:

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

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

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

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

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

Ejemplos:

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

SUPER puntero

SUPER^ puntero:

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

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

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

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

Ejemplo:

- Usando los punteros SUPER y THIS:

Bloque de Función -- FB_Base:

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

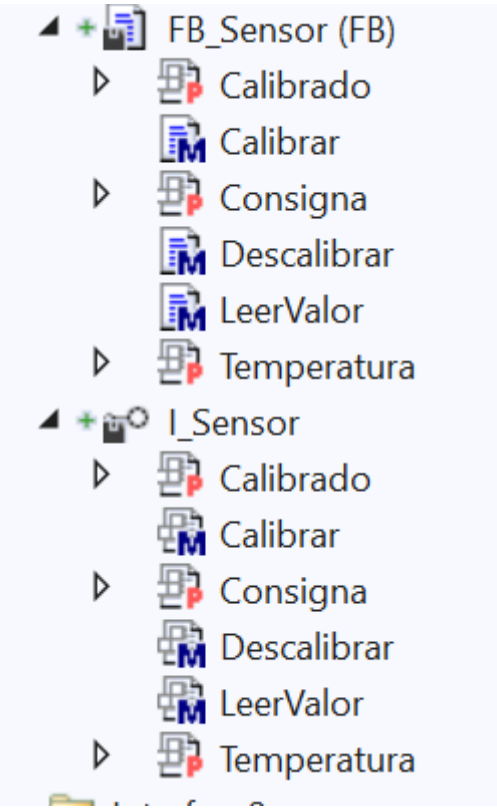
Metodo -- FB_Base.METH_DoIt:

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

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.

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:

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.

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.

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#:

```
1   EmployeeNames = EmployeeList.Where(x=› x.Age › 65)
2                                .Select(x=› x)
3                                .Where(x=› x.YearsOfEmployment › 20)
4                                .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.

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:

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.

```
1   //Syntax:
2    __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 …).

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

- 🔗 __ISVALIDREF, infosys.beckhoff.com

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:

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

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

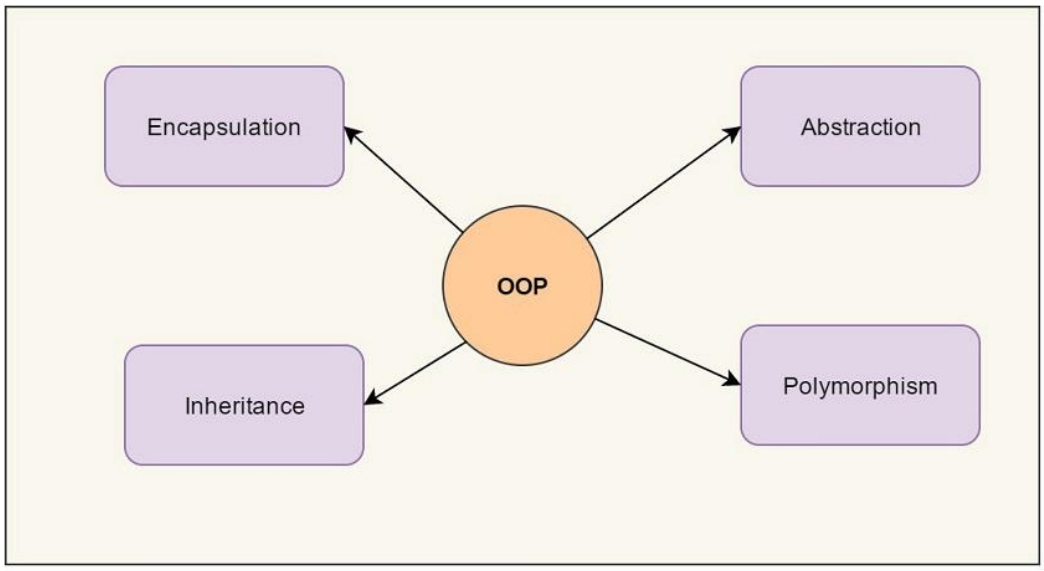
Tabla de Ejemplos:

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!*)

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
- 🔗 Object Oriented I/O (OOIO)

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.

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.

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

---

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:

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.

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
```

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:

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:

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:

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
```

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

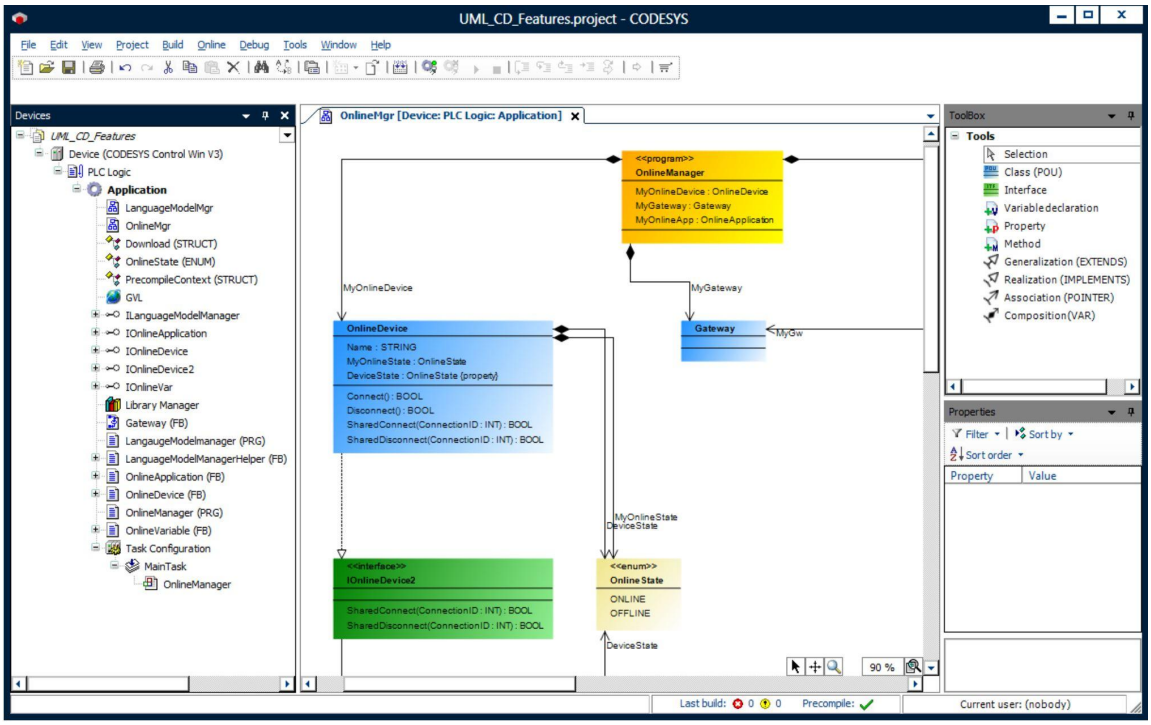
Class UML

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

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.

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.

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

Patrones de Diseño

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

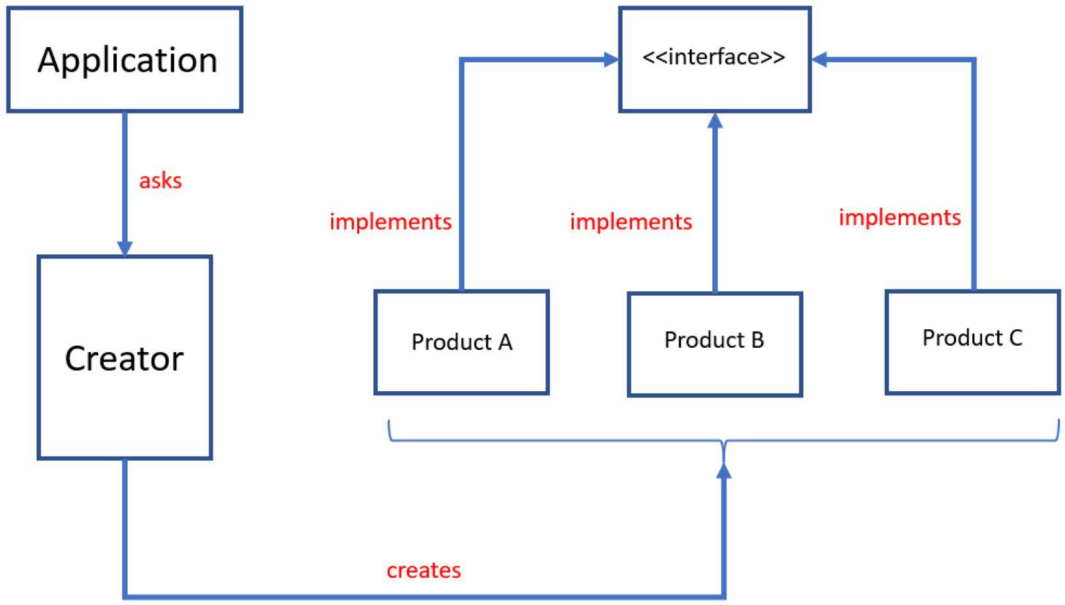
Patron Factory Method

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.

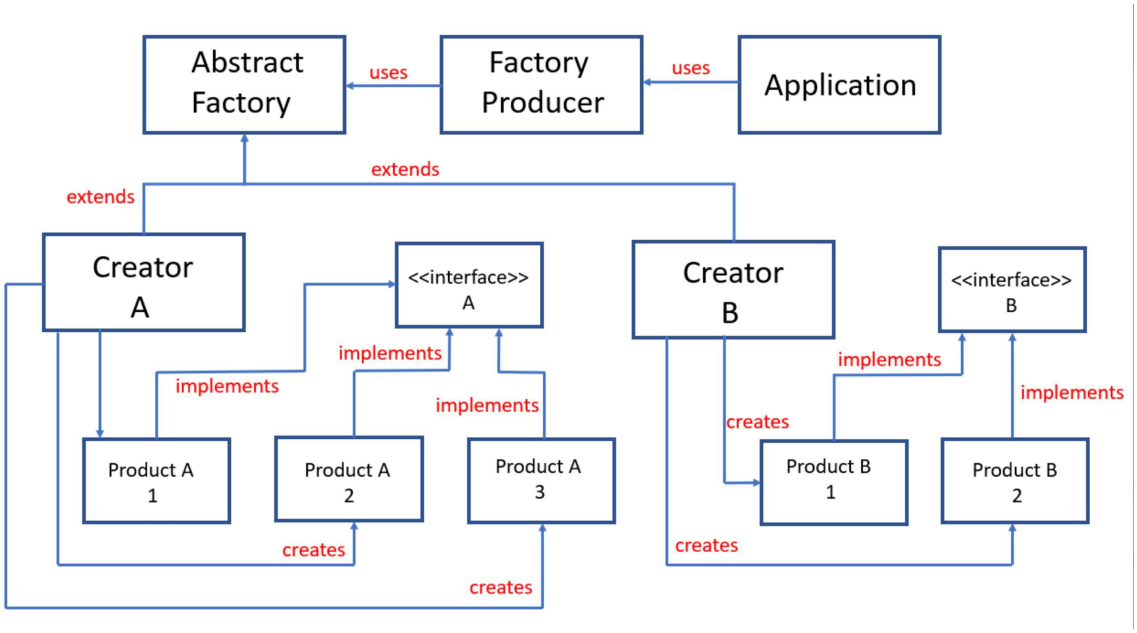
El diagrama de clases UML del método de fábrica es el siguiente:



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.

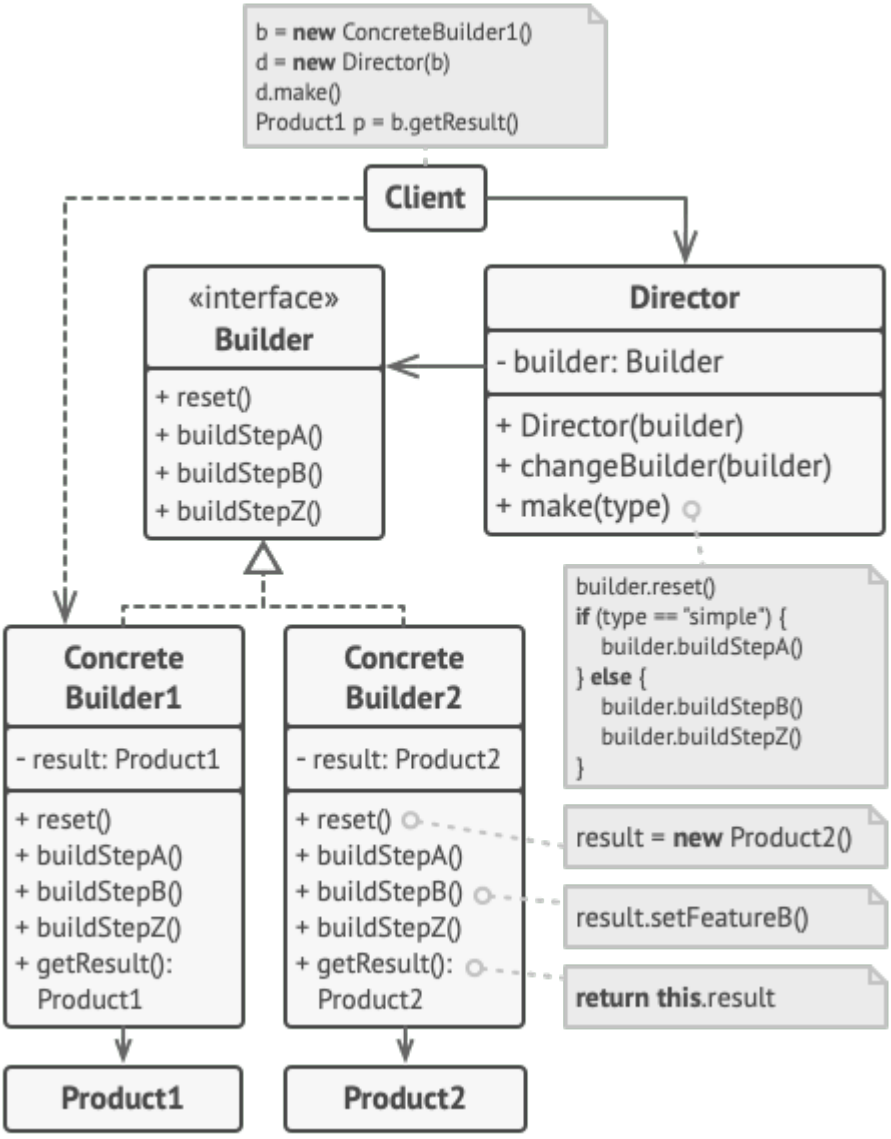
El diagrama de clases UML de la fábrica abstracta es el siguiente:



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:



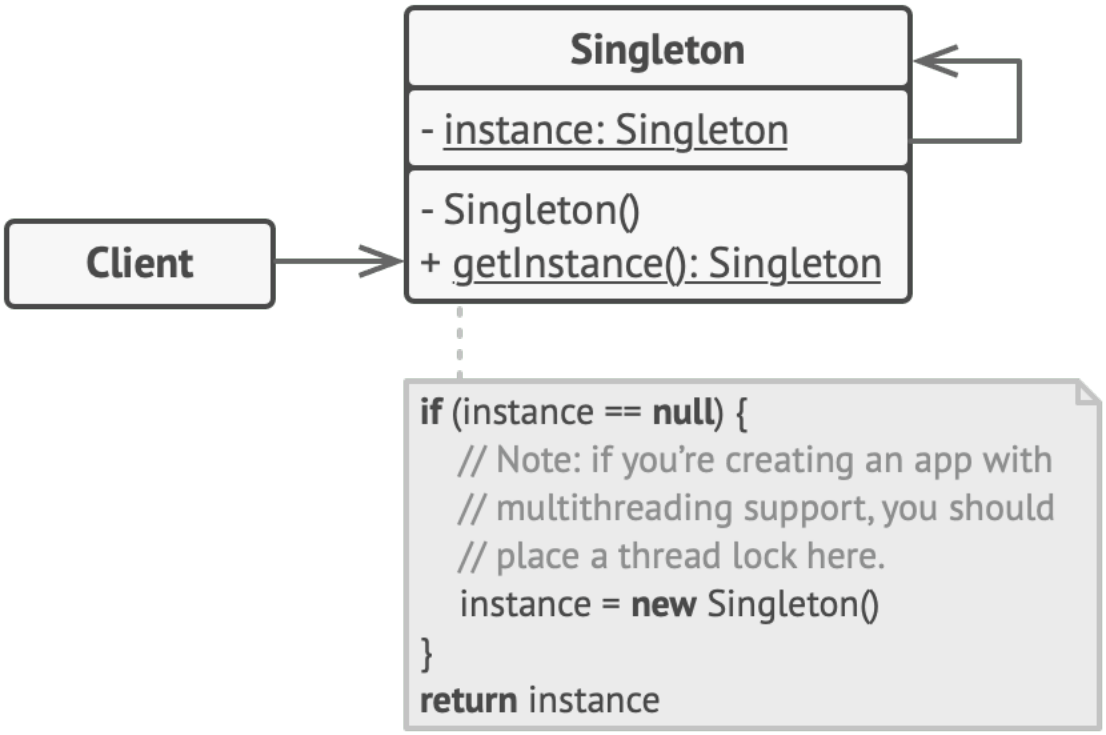
Patron Prototype

- 👉**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 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 necesidad de acoplar la lógica 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.

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 desde cualquier parte del programa.

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.
- Esto es útil cuando solo se necesita una instancia de una clase para coordinar acciones en todo el programa. En resumen, es una forma de controlar la creación de objetos.

Patron Adapter

👉**Adapter** es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

Un adaptador es un patrón para hacer que objetos incompatibles funcionen juntos. La idea es crear una capa intermedia que convierta la interfaz de un objeto en otro. Un nombre alternativo común para el adaptador es contenedor, wrapper (envoltorio)

Ventajas y cuando utilizar:

El patrón de adaptador es útil para integrar código que no funciona bien en conjunto, sin modificar ningún código existente. Este es un ejemplo del principio abierto-cerrado: sin modificar el código fuente existente, los módulos existentes se pueden ampliar.

En el caso más simple, se puede utilizar para métodos de alias; en casos más complejos, se puede utilizar para cambiar por completo el comportamiento de un objeto. Los adaptadores pueden resultar útiles cuando los objetos que deben adaptarse provienen de bibliotecas de terceros que no se pueden modificar.

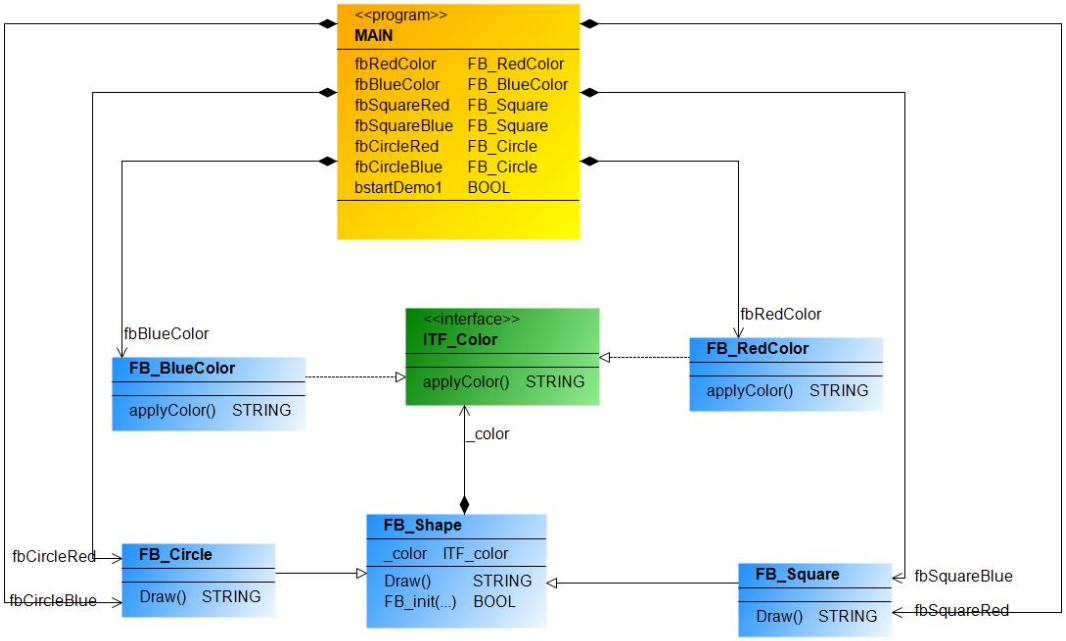
Desventajas:

Puede haber una sobrecarga de rendimiento adicional debido a las operaciones de transformación en el adaptador. Puede complicar la base del código.

Patron Bridge

👉**Bridge** es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

Diagrama de Clases UML - Bridge:



Links de Patron de Diseño - Estructural - Bridge:

- 🔗 refactoring.guru/es/design-patterns/bridge
- 🔗 Bridge Pattern – Design Patterns (ep 11)

Link al Video de Youtube_51:

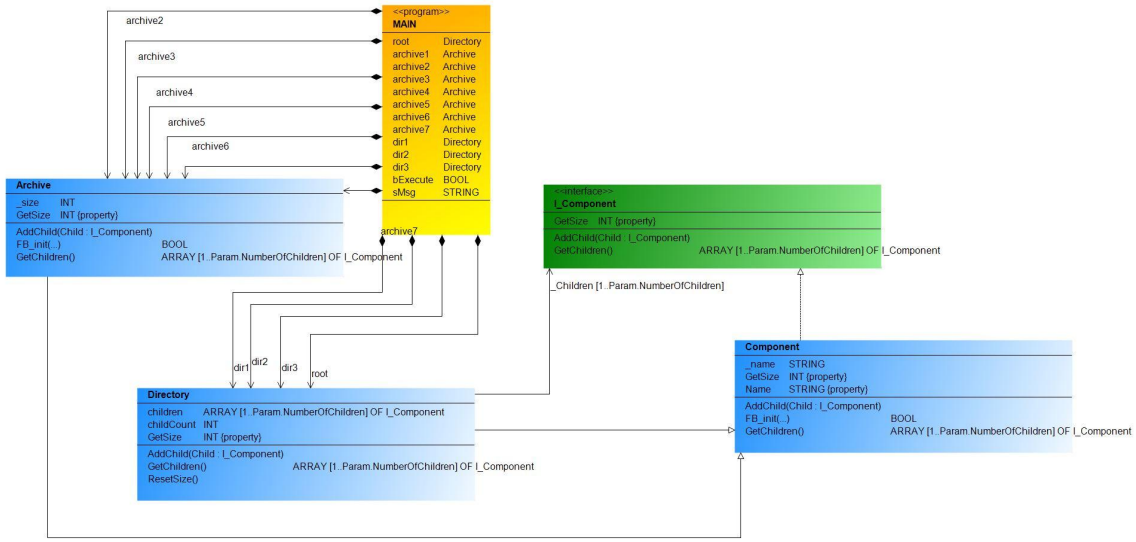
- 🔗 051 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Estructural - Puente

---

Patron Composite

👉**Composite** es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

Diagrama de Clases UML - Composite:



Links de Patron de Diseño - Estructural - Composite:

- 🔗 refactoring.guru/es/design-patterns/composite
- 🔗 Patrón de diseño Composite en C#
- 🔗 Composite Pattern – Design Patterns (ep 14)
- 🔗 Composite Design Pattern – Simple or Complex? This is the question!

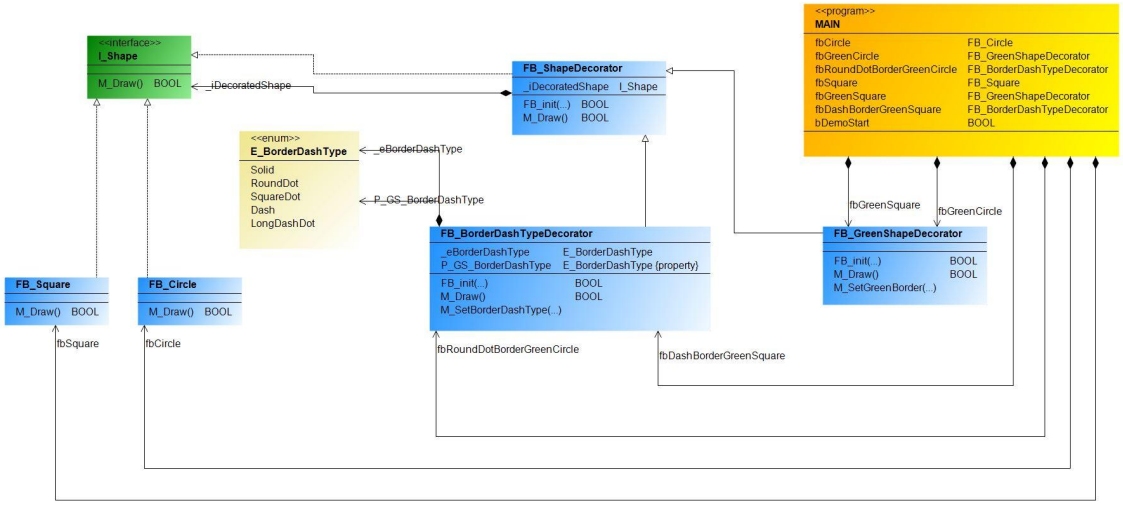
Link al Video de Youtube_52:

- 🔗 052 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Estructural - Compuesto

Patron Decorator

👉**Decorador** es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Diagrama de Clases UML - Decorator:



Links de Patrones de Diseño Decorador:

- 🔗 refactoring.guru/es/design-patterns/decorator
- 🔗 Decorator Design Pattern
- 🔗 github.com,Aliazzzz,Applied-Design-Patterns-in-CODESYS-V3
- 🔗 stefanhenneken.net/IEC 61131-3: The Decorator Pattern
- 🔗 🤩PATRONES de DISEÑO con Typescript - Decorator Pattern | PT 2.

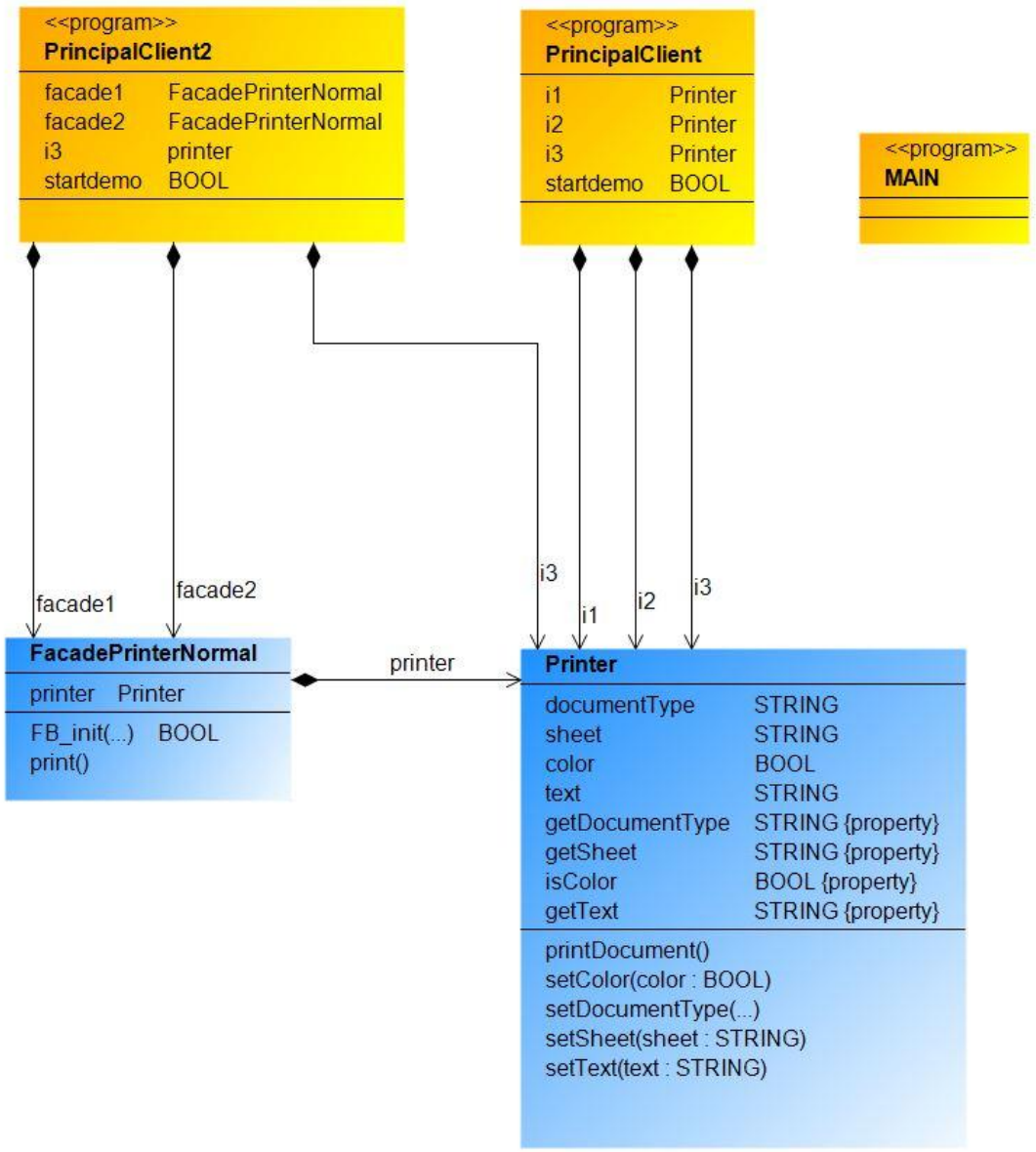
Link al Video de Youtube_53:

- 🔗 053 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Estructural - Decorador

Patron Facade

👉**Facade** es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

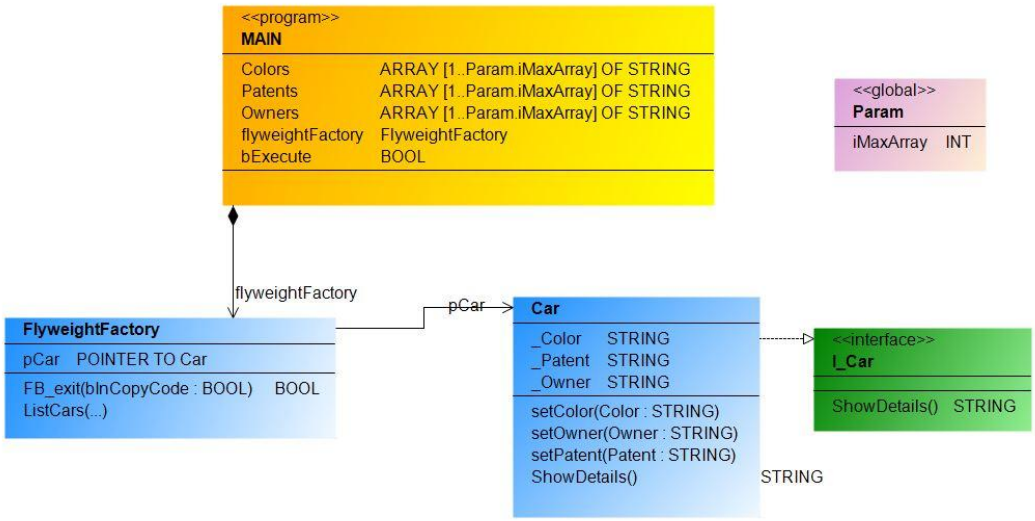
Diagrama de Clases UML - Fachada:



Patron Flyweight

👉**Flyweight** es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

Diagrama de Clases UML - Flyweight:



Links de Patrones de Diseño - Flyweight:

- 🔗 refactoring.guru/es/design-patterns/flyweight
- 🔗 Patrones de diseño de Software – Patrón Flyweight
- 🔗 Patrón de diseño peso ligero

Link al Video de Youtube_55:

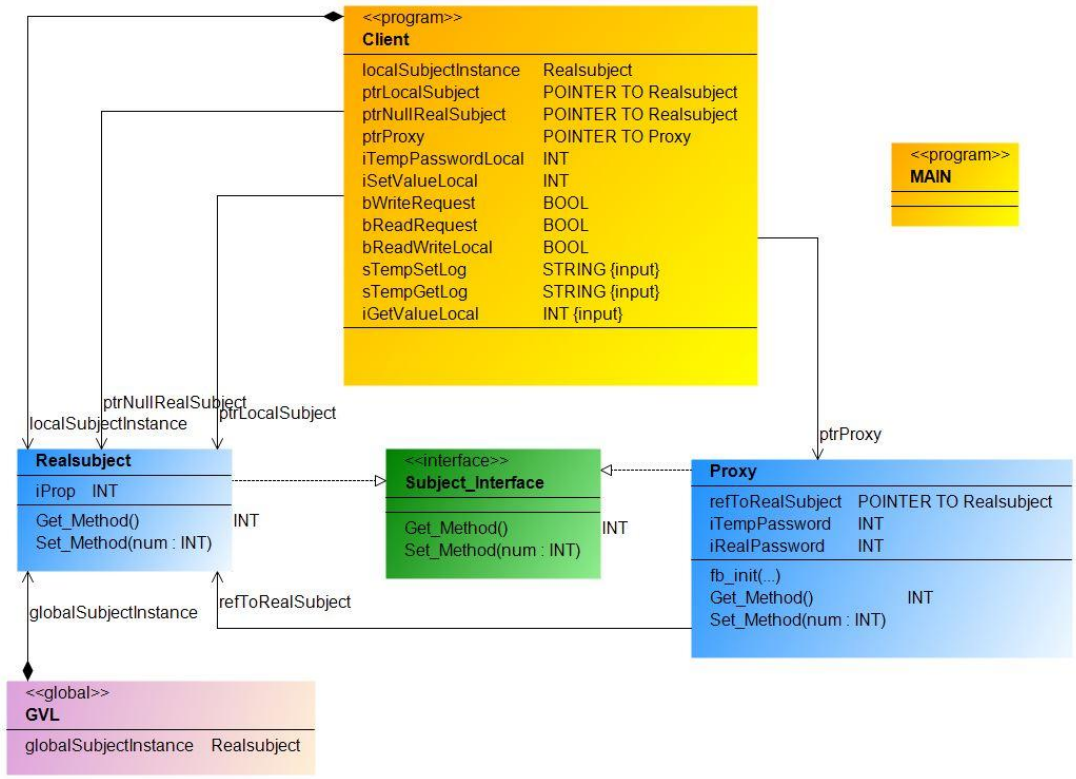
- 🔗 055 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Estructural - Peso Ligero

---

Patron Proxy

👉**Proxy** es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

Diagrama de Clases UML - Proxy:



Links de Patron de Diseño - Structural - Proxy:

- 🔗 refactoring.guru/es/design-patterns/proxy
- 🔗 github.com/Aliazzzz/Applied-Design-Patterns-in-CODESYS-V3/Design Pattern Proxy
- 🔗 Aprende los principales patrones de diseño con .NET y C#. Patron Proxy

---

Patron Chain of Responsibility

- En el 👉**Chain of Responsibility** es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

Links de Patron de Diseño - Comportamiento - Cadena de responsabilidad:

- 🔗 refactoring.guru/design-patterns/chain-of-responsibility
- 🔗 github.com/0w8States/PLC-Design-Patterns/Behavioral_Patterns/Chain_of_Responsibility
- 🔗 Cadena de Responsabilidad I - 30 - Patrones de Diseño de Software

---

Link al Video de Youtube_40:

- 🔗 040 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Comportamiento - Cadena de Responsabilidad

May 12, 2024 19:25:39

May 12, 2024 19:25:39

Patron Command

👉**Command** es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

En el patrón de comando se definen tres capas:

- *Invocador* : Los FB de esta capa activan el comando requerido. El invocador, no conoce el receptor del comando. Pero sabe cómo se inicia un comando.
- *Receptor* : Estos son los FB que representan el correspondiente receptor de los comandos.
- *Comandos* : Cada comando está representado por un FB. Este FB contiene una referencia al receptor. Además, estos comandos tienen un método para activar el comando. Si se llama a este método, el FB de comando sabe qué métodos deben ejecutarse en el receptor para lograr el efecto deseado.

El patrón de comando Se puede ejecutar un comando en un bloque de funciones llamando a un método. El bloque de funciones A llama a un método del bloque de funciones B. Hasta aquí todo bien, pero ¿cómo se pueden intercambiar dichos comandos de manera flexible entre varios bloques de funciones? El patrón de comando proporciona un enfoque interesante.

Definición del patrón de comando La solución al problema es introducir una capa de software. Esta capa encapsula cada comando (con un FB de comando) y contiene todas las llamadas a métodos relevantes para realizar una acción en el dispositivo.

Un FB de comando encapsula una asignación al contener un conjunto de acciones para un determinado receptor. Para ello se combinan las acciones y la referencia del receptor en un FB. A través de cualquier método (por ejemplo, Execute()), el comando FB garantiza que se ejecuten las acciones adecuadas en el receptor. El invocador no ve desde el exterior qué acciones son realmente estas. Solo sabe que cuando llama al método Invoke() , se realizan todos los pasos necesarios.

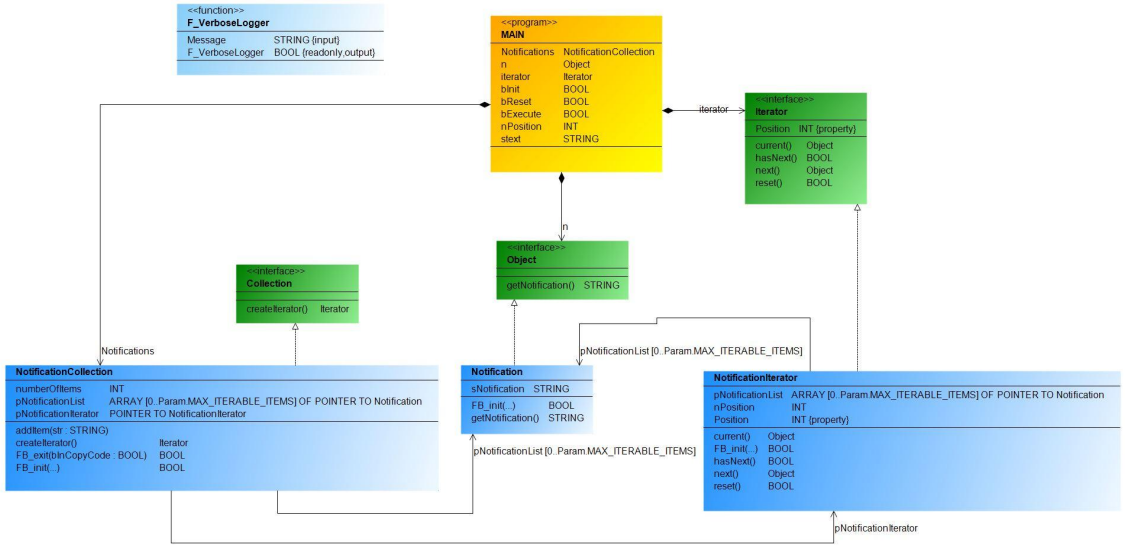
Los beneficios resultantes son bastante convincentes:

- *Desacoplamiento* : El invocador y el receptor están desacoplados entre sí. Como consecuencia, se puede diseñar de forma genérica. El método SetCommand() ofrece la posibilidad de adaptar la asignación de las claves durante el tiempo de ejecución.
- *Ampliabilidad* : Se puede agregar cualquier FB de comando. Incluso si se proporciona una biblioteca, un programador puede definir cualquier comando FB y usarlo, sin necesidad de

Patron Iterator

👉**Iterator** es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

Diagrama de Clases UML - Iterator:



Links de Patron de Diseño - Comportamiento - Iterator:

- 🔗 factoring.guru/design-patterns/iterator
- 🔗 github.com/0w8States/PLC-Design-Patterns/Behavioral_Patterns/Iterator

---

Link al Video de Youtube_42:

- 🔗 042 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Comportamiento - Iterador

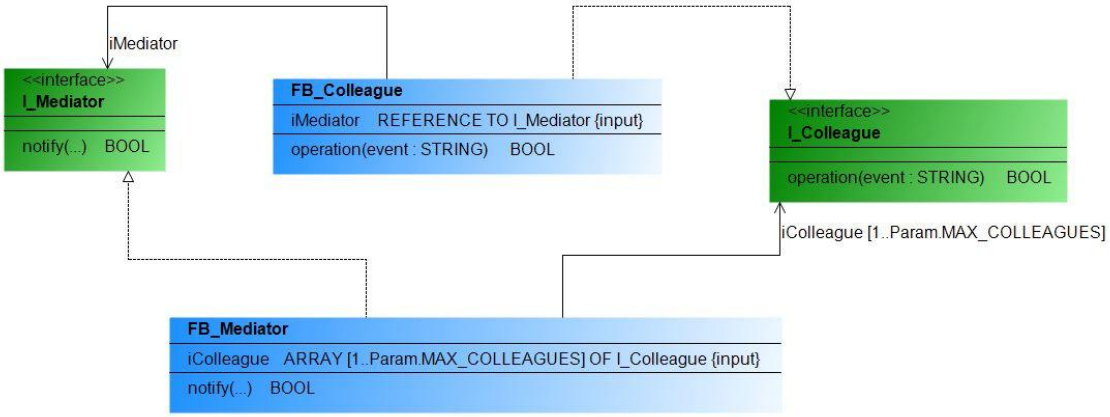
May 12, 2024 19:25:39

May 12, 2024 19:25:39

Patron Mediator

👉**Mediator** es un patrón de diseño de comportamiento que te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

Diagrama de Clases UML - Mediator:



Links de Patron de Diseño - Comportamiento - Mediator:

- 🔗 refactoring.guru/design-patterns/mediator
- 🔗 github.com/0w8States/PLC-Design-Patterns/Behavioral_Patterns/Mediator

---

Link al Video de Youtube_43:

- 🔗 043 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Comportamiento - Mediador

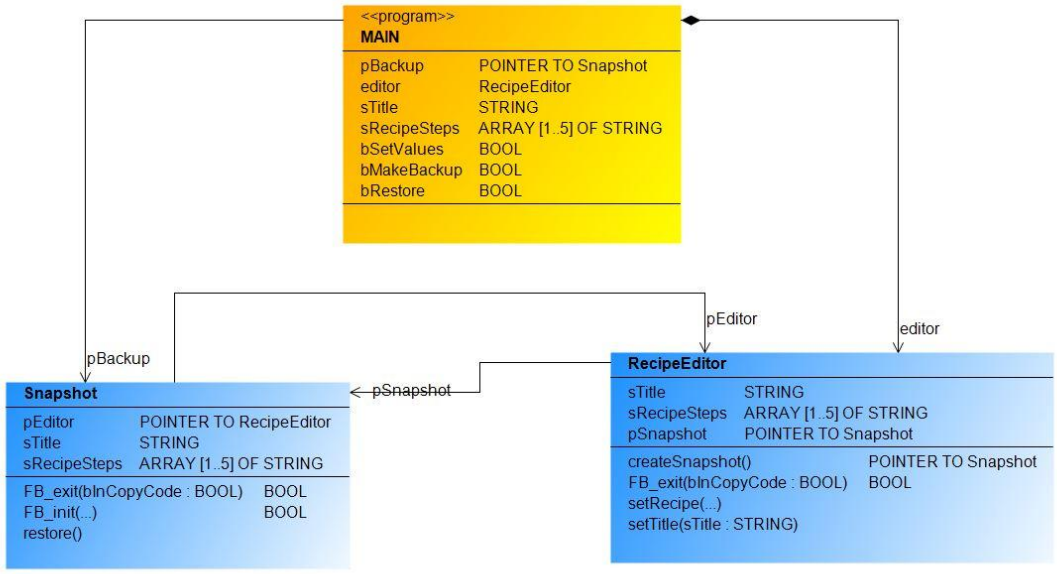
May 12, 2024 19:25:39

May 12, 2024 19:25:39

Patron Memento

👉**Memento** es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

Diagrama de Clases UML - Memento:



Links de Patron de Diseño - Comportamiento - Memento:

- 🔗 refactoring.guru/design-patterns/memento
- 🔗 github.com/0w8States/PLC-Design-Patterns/Behavioral_Patterns/Momento

---

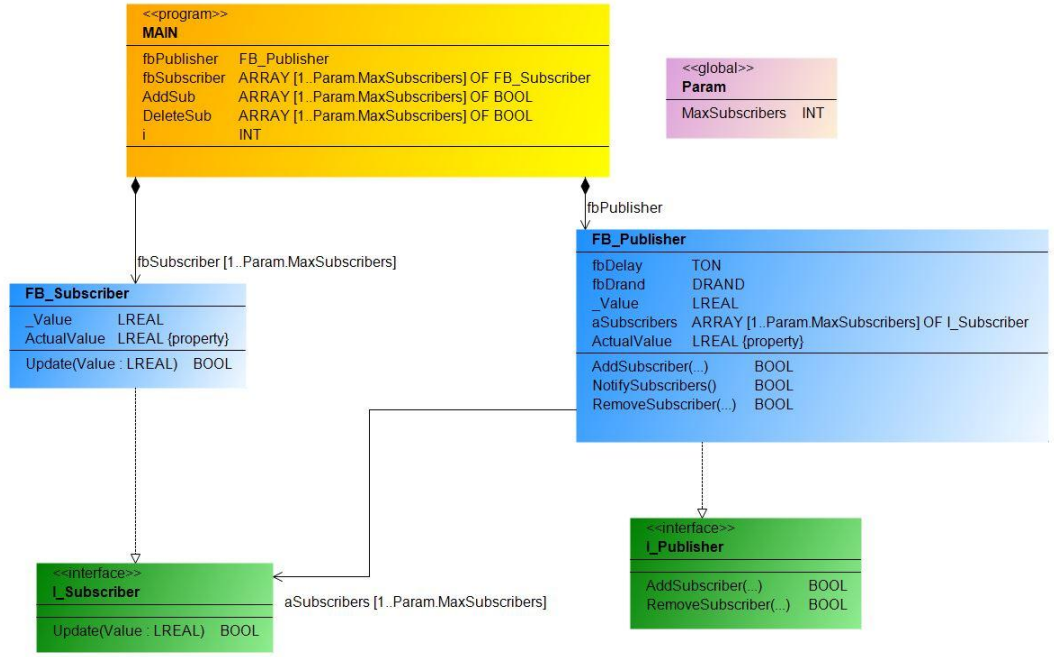
Link al Video de Youtube_44:

- 🔗 044 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Comportamiento - Recuerdo

Patron Observer

👉**Observer** es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

Diagrama de Clases UML - Observer:



Links de Patron de Diseño - Comportamiento - Observer:

- 🔗 refactoring.guru/design-patterns/observer
- 🔗 twincontrols.com, Observer Design Pattern
- 🔗 github.com/Aliazzzz/Applied-Design-Patterns-in-CODESYS-V3/Observer
- 🔗 github.com/0w8States/PLC-Design-PatternsBehavioral_Patterns/Observer
- 🔗 github.com/mcclureTC/TcObserverPattern

Patron State

- 👉**State** es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.
- El patrón de diseño de Estados es muy similar a lo que se conoce como (FSM) Máquina de Estados Finitos .
- 👉PLC-Design-Patterns/State/README.md

Links de Patron de Diseño - Comportamiento - State:

- 🔗 refactoring.guru/design-patterns/state
- 🔗 github.com/0w8States/PLC-Design-Patterns/Behavioral_Patterns/State

---

Link al Video de Youtube_46:

- 🔗 046 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Comportamiento - Estado

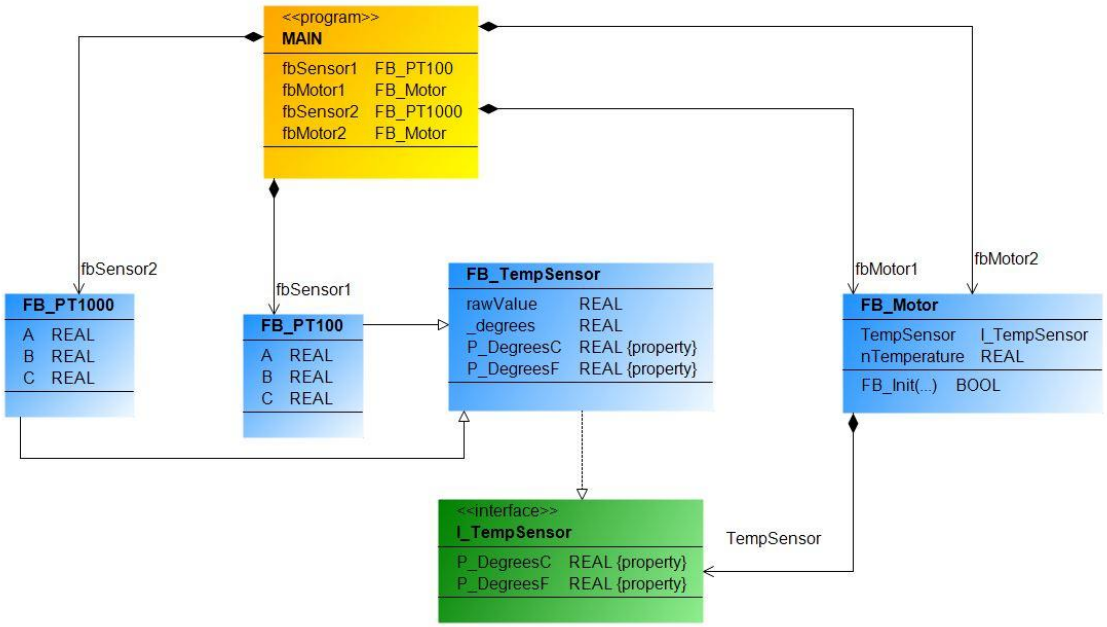
May 12, 2024 19:25:39

May 12, 2024 19:25:39

Patron Strategy

👉**Strategy** es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

Diagrama de Clases UML - Strategy:



Links de Patron de Diseño - Comportamiento - Strategy:

- 🔗 refactoring.guru/gn-patterns/strategy
- 🔗 TwinCAT with Head First Design Patterns Ch.1 - Intro/Strategy Pattern
- 🔗 ¿PATRONES de DISEÑO en PROGRAMACIÓN FUNCIONAL?
- 🔗 github.com/0w8States/PLC-Design-Patterns/Behavioral_Patterns/Strategy

---

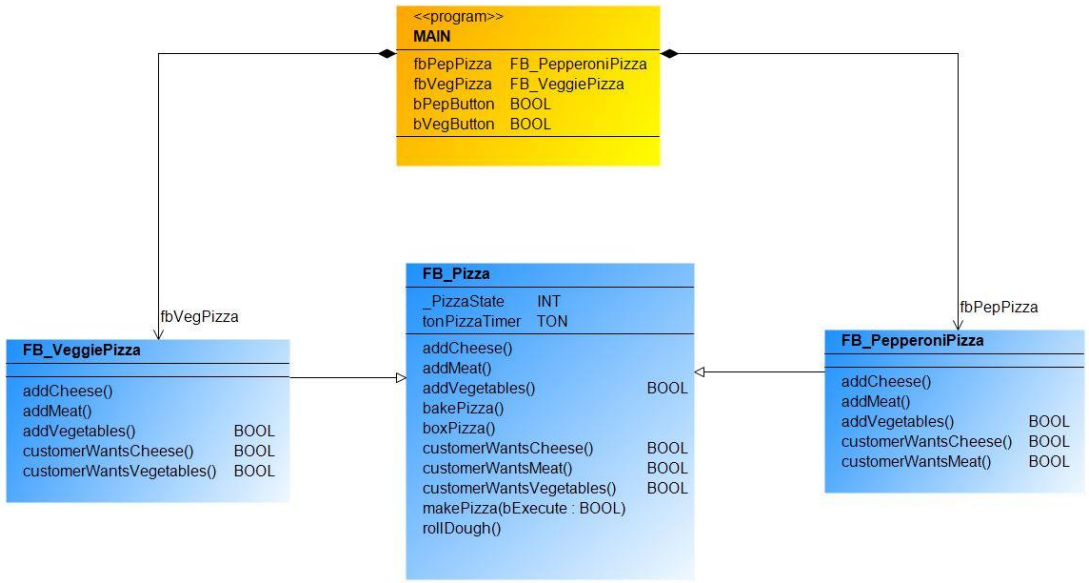
Link al Video de Youtube_47:

- 🔗 047 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Comportamiento - Estrategia

Patron Template Method

👉**Template Method** es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

Diagrama de Clases UML - Template Method:



Links de Patron de Diseño - Comportamiento - Template Method:

- 🔗 refactoring.guru/design-patterns/template-method
- 🔗 github.com/0w8States/PLC-Design-Patterns/Behavioral_Patterns/Template_Method

---

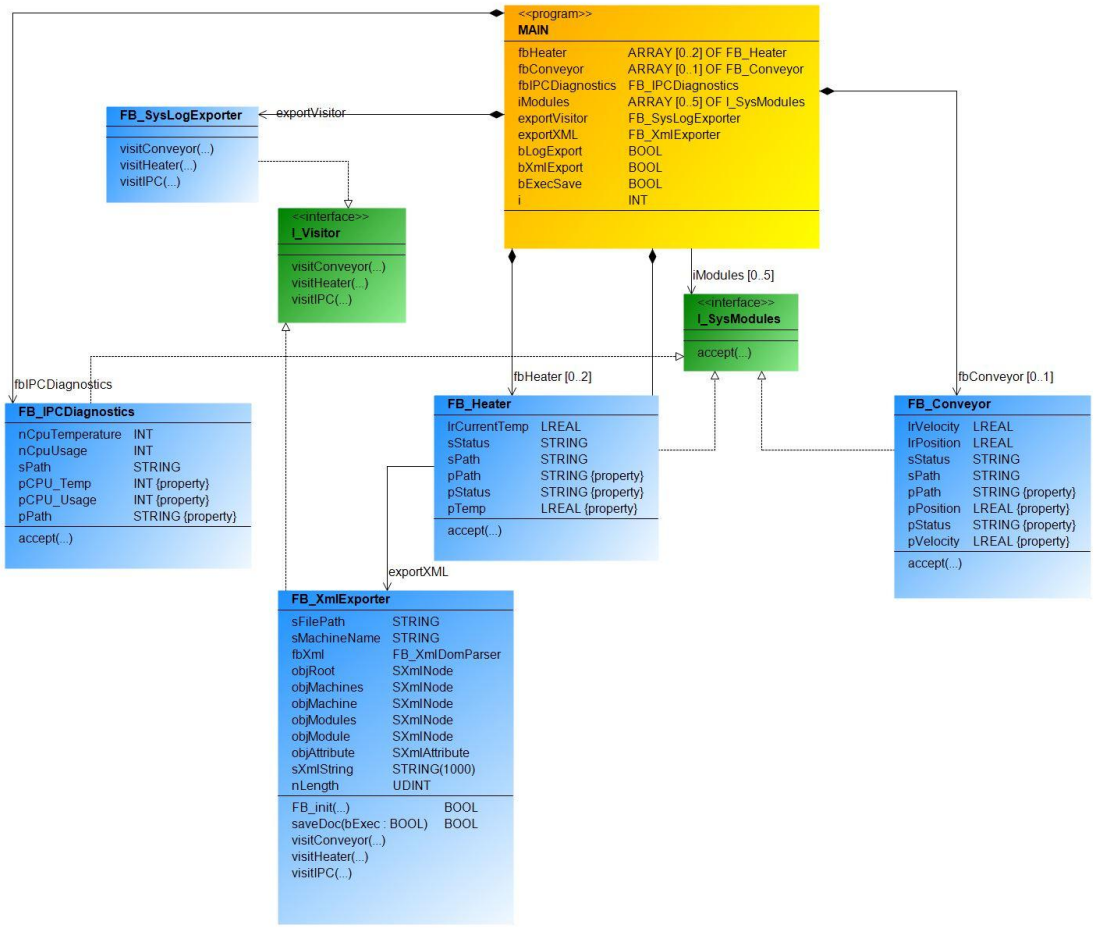
Link al Video de Youtube_48:

- 🔗 048 - OOP IEC 61131-3 PLC -- Patrones de Diseño - Comportamiento - Metodo de plantilla

Patron Visitor

Visitor es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.

Diagrama de Clases UML - Visitor:



Links de Patron de Diseño - Comportamiento - Visitor:

- 🔗refactoring.guru/design-patterns/visitor
- 🔗github.com/0w8States/PLC-Design-Patterns/Visitor
- 🔗Design Patterns - The Visitor Pattern

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, este concepto de gestión de bibliotecas es bastante común.

¿Cuáles son las ventajas de usar las bibliotecas (Librerias)?

- 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.

Ajustes de Configuración en el IDE Visual Studio XAE para GIT:

- 1 -- Tools --> Options --> Source Control --> Plug-in Selection --> Current source control plug-in: Git
- 2 -- Tools --> Options --> TwinCAT --> XAE Environment --> File settings -- enable all to: True
- 3 -- Tools --> Options --> TwinCAT --> PLC Environment --> Write Options -- Separate LineIDs : True
    - 🔗 s-linedid

Ajustes de Configuración en el IDE Visual Studio XAE Opcionales:

- 1 -- (Optional): Tools --> Options --> TwinCAT --> PLC Environment --> Text Editor --> Text area --> End of line markers : Enable
- 2 -- (Optional): Tools --> Options --> TwinCAT --> PLC Environment --> Smart coding -- Declared unknow variables automatically/(AutoDeclare) : Deselect

---

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 🔴

May 12, 2024 19:25:39

May 12, 2024 19:25:39

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

May 12, 2024 19:25:39

May 12, 2024 19:25:39

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.

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

May 12, 2024 19:25:39

May 12, 2024 19:25:39

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.

May 12, 2024 19:25:39

May 12, 2024 19:25:39

Simulacion

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

May 12, 2024 19:25:39

May 12, 2024 19:25:39

Links OOP

Links de OOP:

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

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

May 12, 2024 19:25:39

May 12, 2024 19:25:39