Requeriments

👨‍🎓 Requirements 👩‍🎓 :

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:

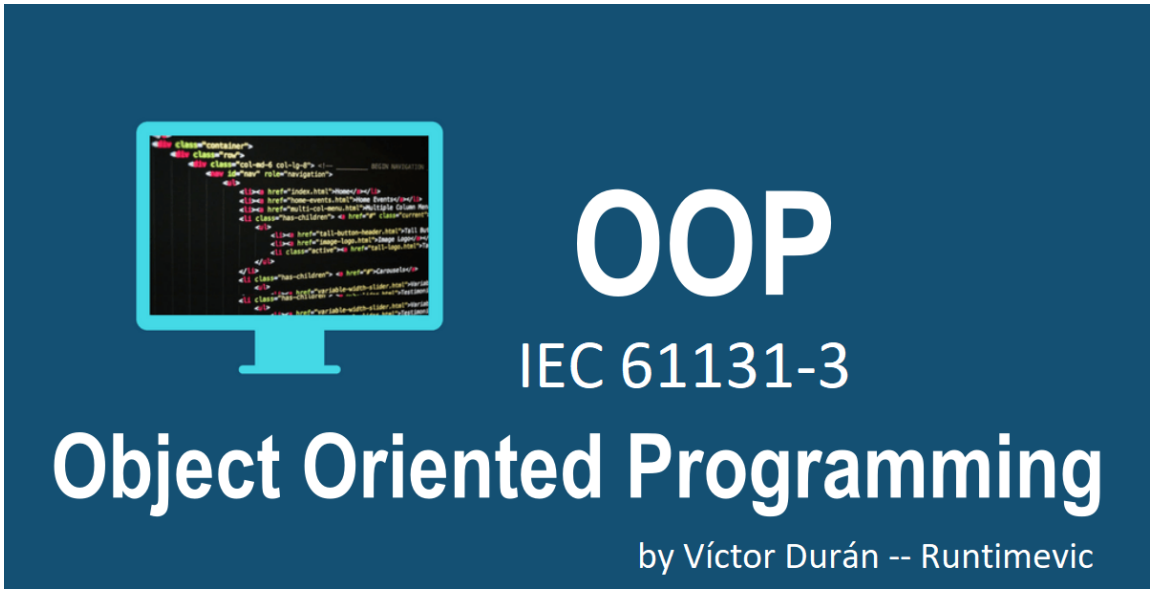
- Clonar el repositorio de 🔗GitHub:

  $ 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...

📖 Curso Programación Orientada a Objetos Youtube -- OOP :



by Runtimevic -- Víctor Durán Muñoz.

¿ Qué es OOP?

- Es un paradigma que hace uso de los objetos para la construcción de los software.

  . ¿ Qué es un paradigma?

  - Tiene diferentes interpretaciones, puede ser un **modelo**, **ejemplo** o **patrón.**
  - Es una **forma** o un **estilo** de programar.
  - se busca plasmar la realidad hacia el código.

. ¿Cómo pensar en Objetos?

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

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

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

```
🚗 Ejemplo: (Coche)

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

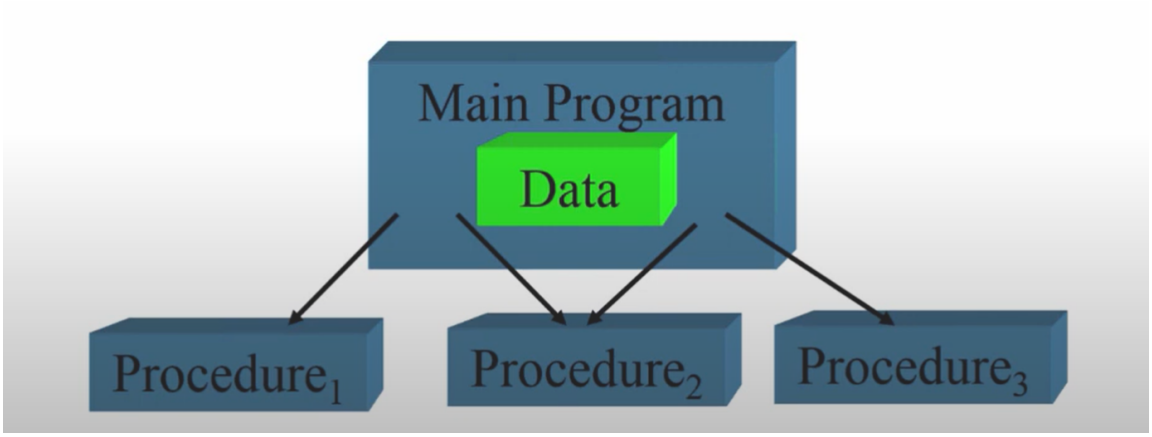
🔗 Codesys admite OOP

🔗 Beckhoff TwinCAT 3 admite OOP

Types of paradigms

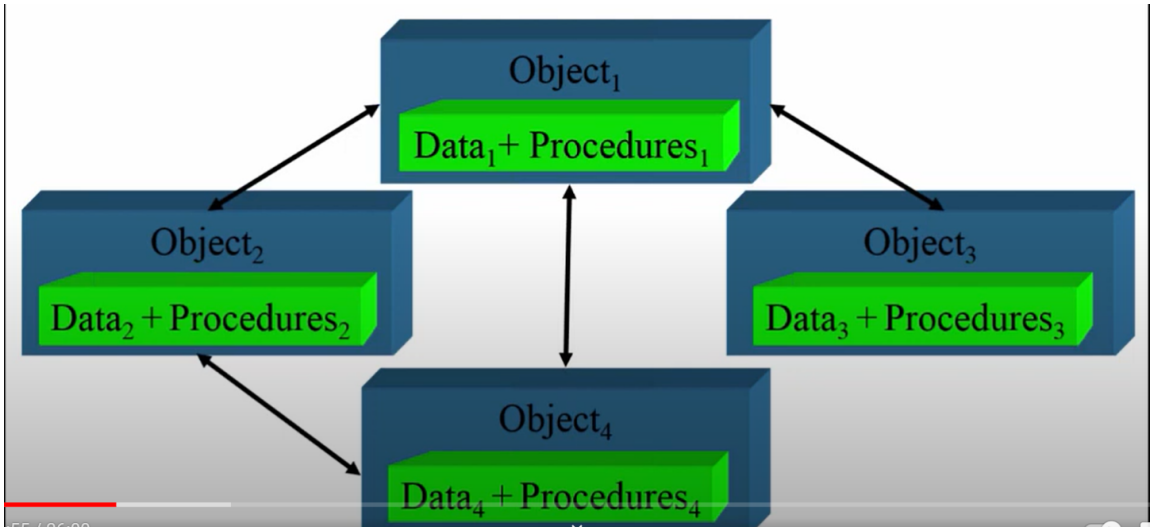
. Tipos de paradigmas:

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



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

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

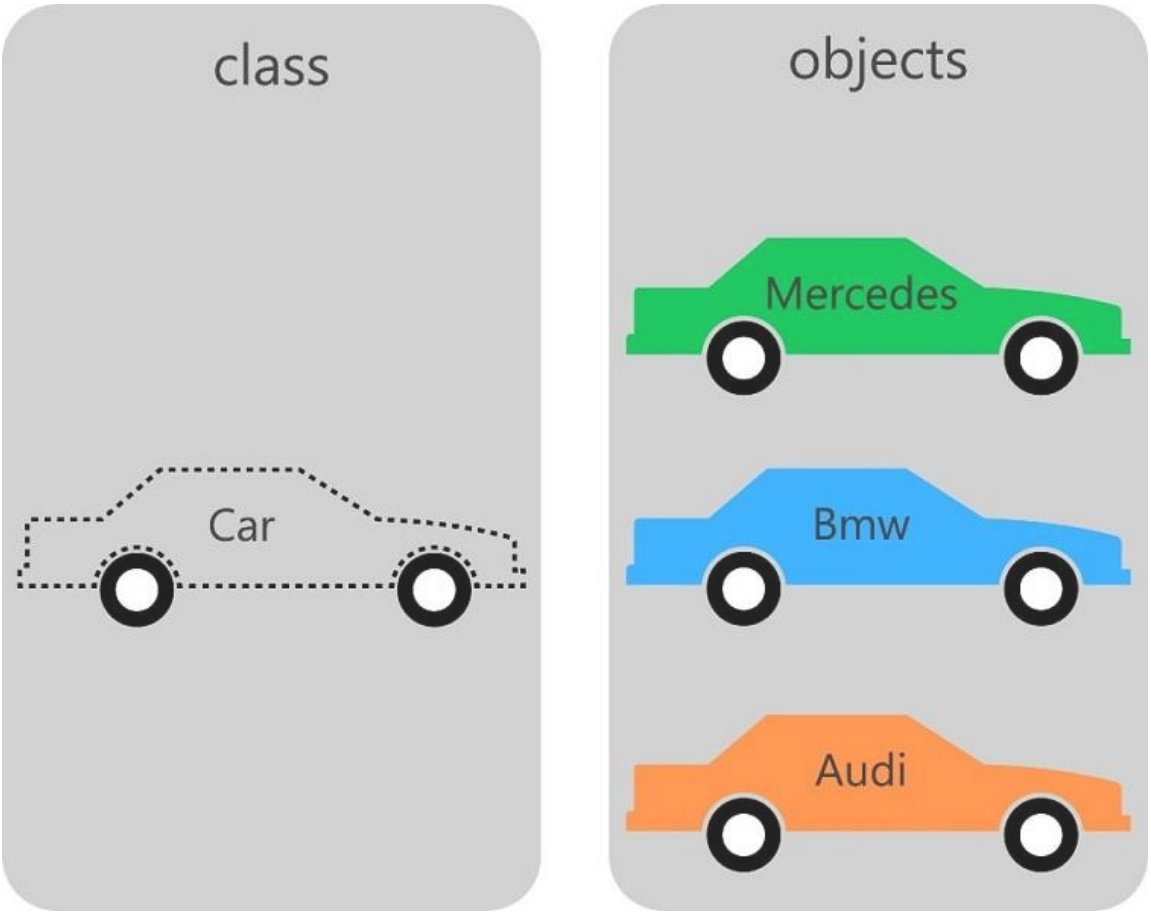


Ventajas de la Programación OOP:

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

. 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 IEC61131-3

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

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

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

    Color : STRING;
    Marca : STRING;

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

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

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

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

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

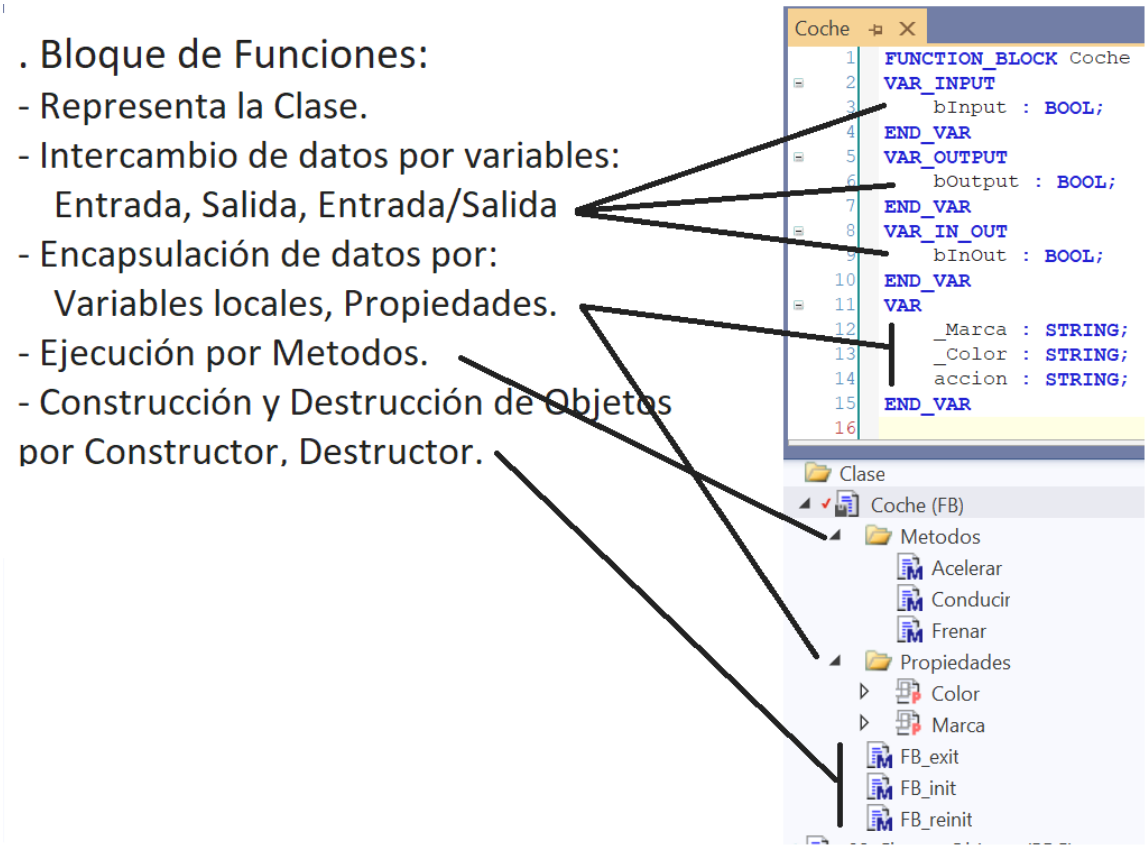
- 🔗 methods-properties-and-inheritance (stefanhenneken)

Function Block

Declaracion de un Function Block:

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

Implementación Bloque de Funciones:



Constructor and Destructor

Métodos 'FB_Init', 'FB_Reinit' y 'FB_Exit'

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

Access modifiers

modificadores de acceso:

- public: son accesibles luego de instanciar la clase.
- private: son accesibles dentro de la clase.
- protected: son accesibles a través de la herencia.
- internal:

- PUBLIC: Corresponds to the specification of no access modifier
- PRIVATE: Access to the method is restricted to the function block or the program respectively.
- PROTECTED: Access to the method is restricted to the program or the function block and its derivatives respectively.
- INTERNAL: Access to the method is limited to the namespace (the library). In addition to these access modifiers, you can manually add the FINAL modifier to a method:
- FINAL: Overwriting the method in a derivative of the function block is not allowed. This means that the method may not be overwritten/extended in a possibly existing subclass.

Method

metodos

Los métodos dividen la clase o el 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) 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 propósitos en lugar de las variables que manipulan, mejorando así la lectura de código y 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 variable global lista).

Especificadores de acceso para los Metodos:

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

- PÚBLICO - Cualquiera puede llamar al método, no hay restricciones.
- PRIVADO - El método está disponible solo dentro de la POU. No se puede llamar desde fuera de la POU.

- PROTEGIDO- Solo su propia POU o las POU derivadas de ella pueden acceder al método. La derivación se analiza a continuación.
- INTERNO- 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.
- FINAL -El método no puede ser sobrescrito por otro método. La sobrescritura de métodos se describe a continuación.

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

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...

🔗Documentación Codesys del Objeto método

🔗Documentación de Beckhoff del Objeto método

Property

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.

Especificadores de acceso:

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

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

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

En el Objeto Propiedad es obligatorio que retorne un valor...

🔗Documentación de Codesys del Objeto propiedad

🔗Documentación de Beckhoff del Objeto propiedad

- 🔗 https://twincontrols.com/community/twincat-troubleshooting/utilizing-properties/#post-76

Inheritance

Herencia:

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 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 PROTEGIDOS , el bloque de funciones secundario (FB_NewEngine) podrá acceder a ellos, pero no desde fuera de FB_NewEngine .

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

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

PÚBLICO Cualquiera puede llamar o crear una instancia de la POU. Además, si la POU es un FUNCTION_BLOCK , se puede usar para la herencia. No se aplican restricciones. INTERNO La POU solo se puede utilizar dentro de su propio espacio de nombres. Esto permite que las POU estén disponibles solo dentro de una determinada biblioteca, por ejemplo. FINAL El FUNCTION_BLOCK no puede servir como un bloque de funciones principal. Los métodos y las propiedades de esta POU no se pueden heredar. FINAL solo está permitido para POU del tipo FUNCTION_BLOCK . La configuración predeterminada donde no se define ningún especificador de acceso es PUBLIC . Los especificadores de acceso PRIVATE y PROTECTED no están permitidos en las declaraciones de POU.

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

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

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

1 2 3 4 5 6 7 8 9 10 11 12 METHOD PUBLIC Start VAR_INPUT nGear : INT := 2; fVelocity : LREAL := 8.0; END_VAR

IF (fVelocity < MaxVelocity) THEN velocityInternal := fVelocity; ELSE velocityInternal := MaxVelocity; END_IF Gear := nGear; // new La línea 12 copia el parámetro nGear a la propiedad Gear .

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

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

imagen05

1 fbNewEngine.Start(1, 7.5); llama al método Start() en FB_NewEngine , ya que este método ha sido redefinido (sobrescrito) en FB_NewEngine .

Mientras que

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

Interface

interface:

Una interfaz es una clase que contiene métodos y propiedades sin implementación. La interfaz se puede implementar en cualquier clase, pero esa clase debe implementar todos sus métodos. y propiedades.

Si bien la herencia es una relación "es un", las interfaces se pueden describir como "se comporta como" o “tiene una” relación.

Las interfaces son objetos que permiten que varias clases diferentes tengan algo en común con menos dependencias. Las clases y los bloques de funciones pueden implementar varias interfaces diferentes. Uno puede pensar en los métodos y propiedades de la interfaz como acciones que significan cosas diferentes dependiendo de quién los esté ejecutando. Por ejemplo, la palabra "Correr" significa "mover a una velocidad más rápido que un paseo” para un ser humano, pero significa “ejecutar” para las computadoras.

Las clases o bloques de funciones que no comparten similitudes pueden implementar la misma interfaz. En este caso, la implementación de los métodos en cada clase puede ser totalmente diferente. Esto abre muchos enfoques de programación poderosos:

- Las POU pueden llamar a una interfaz para ejecutar un método o acceder a una propiedad, sin saber cuál clase o FB con el que se trata o cómo va a ejecutar la operación. La interfaz luego apunta a una clase o bloque de función que implementa la interfaz y la operación es ejecutado;
- Los programadores pueden crear cajas de interruptores fácilmente personalizables usando polimorfismo.

🔗Codesys Comando 'Implementar interfaces'

🔗Codesys Objeto Interface

🔗Codesys Implementando Interfaces

🔗Beckhoff Objeto Interface

🔗Beckhoff Implementando Interfaces

Variable types and special variables

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
- Input Variables - VAR_INPUT
- Output Variables - VAR_OUTPUT
- Input/Output Variables - VAR_IN_OUT, VAR_IN_OUT CONSTANT
- Global Variables - VAR_GLOBAL
- 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

- 🔗 infosys.beckhoff.com/
- 🔗 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

SUPER pointer

SUPER puntero:

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

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

- SUPER puntero Infosys Beckhoff

THIS pointer

ESTE puntero:

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

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

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

- THIS puntero Infosys Beckhoff

Types of data

tipos de datos:

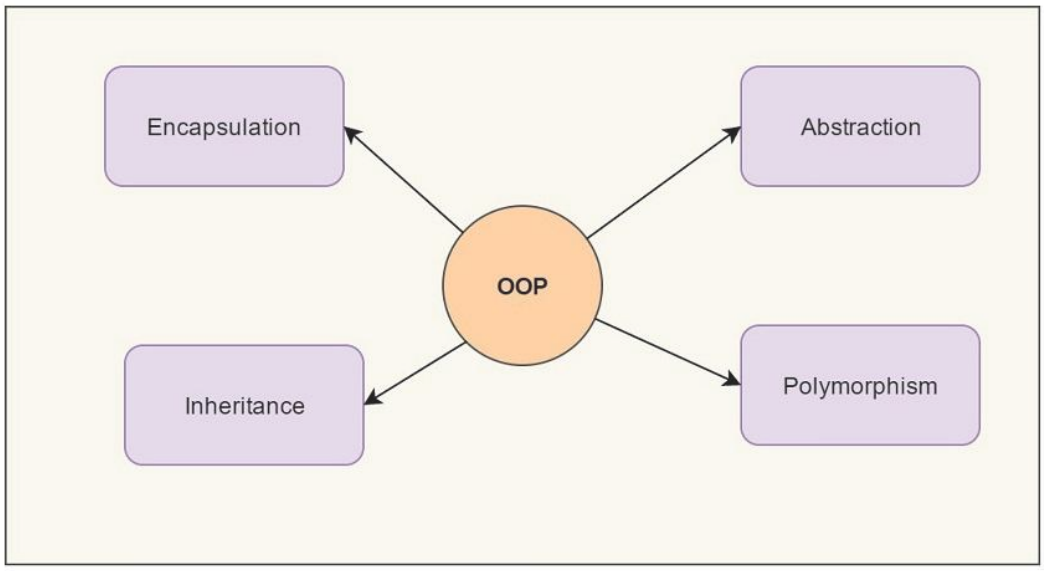
Las ventajas de las estructuras de datos.

- 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)
- 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.

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

Abstraction

abstraccion

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

- https://www.plccoder.com/abstract/

Encapsulation

encapsulamiento

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

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

Inheritance

herencia

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

Polymorphism

polimorfismo

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

- https://www.plccoder.com/polymorphism/
- https://www.plccoder.com/abstract/

¿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:

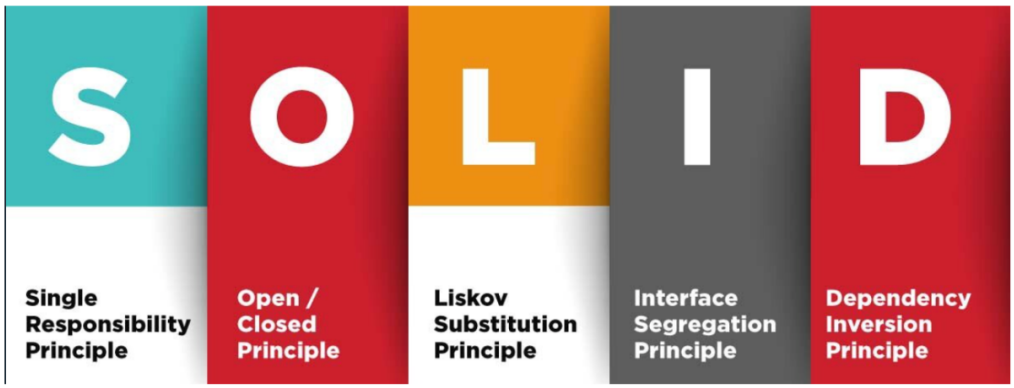
Clases abstractas	**Interfaces**
1.- Limitadas a una sola implementación.	1. No tiene limitación de implementación.

Clases abstractas	Interfaces
2.- Pueden definir comportamiento base.	2. Expone propiedades y métodos abstractos (sin lógica).

.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**.



Principles of Object Oriented Design

- Propuesta por **Robert C.Martin** en el 2000. - Son **recomendaciones** para escribir un código **sostenible,mantenible,escalable y robusto**. - Beneficios: - Alta **Cohesión**. Colaboracion entre clases. - Bajo **Acoplamiento**. Evitar que una clase dependa fuertemente de otra clase.

- Principio de Responsabilidad Única: Una clase debe tener **una razón** para existir mas no para cambiar.
- Principio de Abierto/Cerrado: Las piezas del software deben estar **abiertas para la extensión** pero **cerradas para la modificación**.
- Principio de Sustitución de Liskov: Las **clases subtipos** deberían ser reemplazables por sus **clases padres**.
- Principio de Segregación de Interfaz: Varias **interfaces** funcionan **mejor que una sola**.
- Principio de Inversión de Dependencia: Clases de **alto nivel** no deben depender de las clases **bajo nivel**.

Sole Responsibility Principle

Principio de Responsabilidad Única

Open/Closed Principle

Principio de Abierto/Cerrado

Liskov Substitution Principle

Principio de sustitución de Liskov

Interface Segregation Principle

Principio de Segregación de Interfaz

Dependency Inversion Principle

Principio de Inversión de Dependencia

UML

UML

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

Class UML

Class UML

UML:

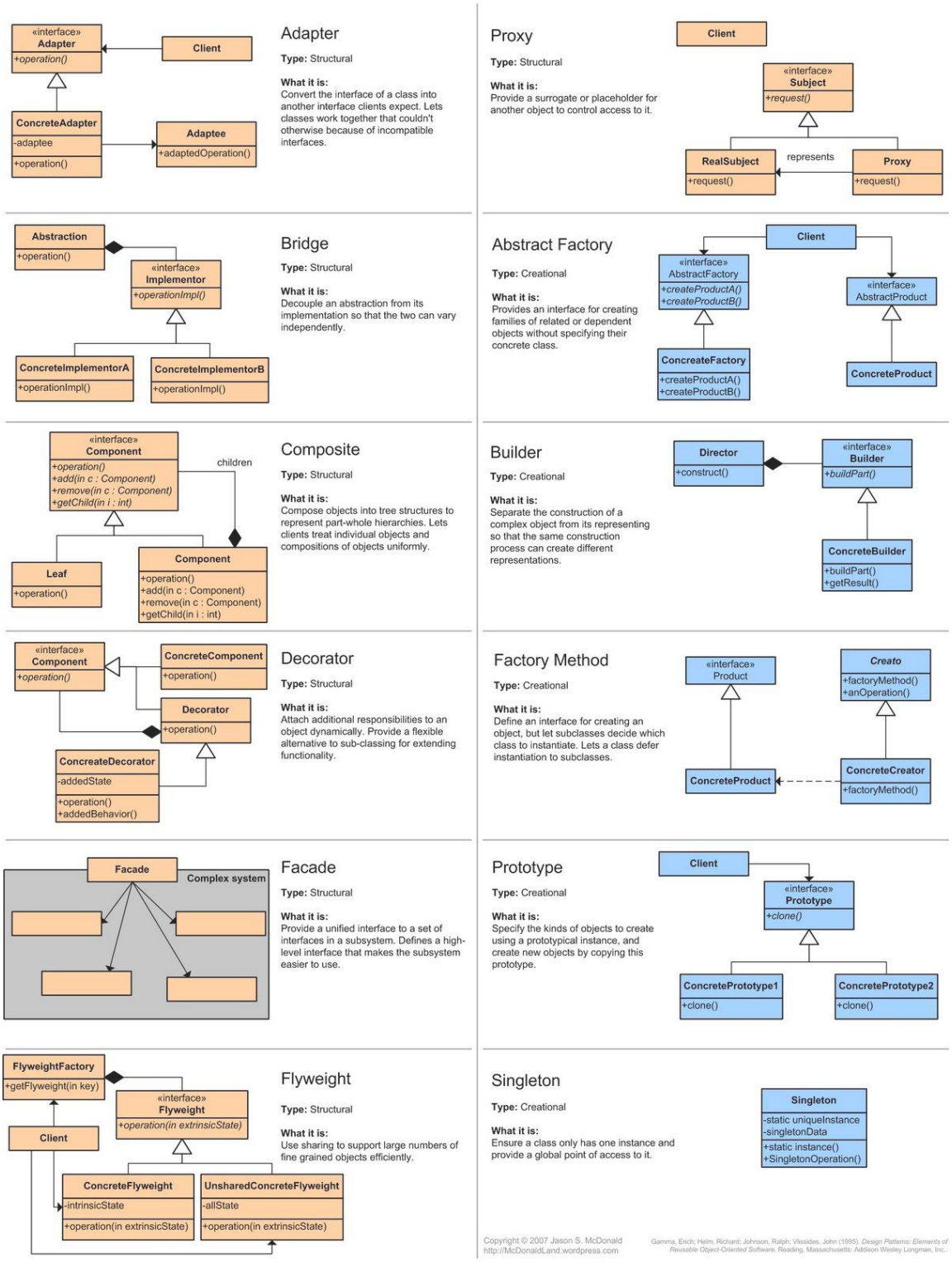
- www.lucidchart.com/tutorial-de-diagrama-de-clases-uml
- www.edrawsoft.com/uml-class-diagram-explained
- blog.visual-paradigm.com/what-are-the-six-types-of-relationships-in-uml-class-diagrams
- Ingeniería del Software: Fundamentos de UML usando Papyrus
- plantuml.com/class-diagram
- www.planttext.com

StateChart UML

state chart:

PATRONES DE DISEÑOS

“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



The Catalog of Design Patterns

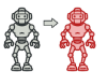
Creational patterns

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

Factory Method

Abstract Factory

Builder

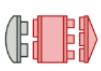


Prototype

Singleton

Structural patterns

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



Adapter

Bridge

Composite

Decorator

Facade

Flyweight

Proxy

Behavioral patterns

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

Chain of Responsibility

Command

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

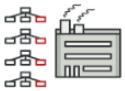
Visitor

Patrones de diseño creacional

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

Método de fábrica

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

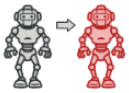


Fábrica abstracta

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

Constructor

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



Prototipo

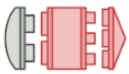
Le permite copiar objetos existentes sin hacer que su código dependa de sus clases.

único

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

Patrones de diseño estructural

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



Adaptador

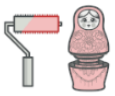
Permite que objetos con interfaces incompatibles colaboren.

Puente

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

Compuesto

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

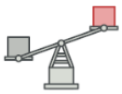


Decorador

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

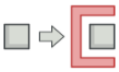
Fachada

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



peso mosca

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



Apoderado

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

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

- Creacionales.
- Estructurales.
- De comportamiento.

Clasificación según su ámbito:

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

Patrones creacionales:

- Builder
- Singleton
- Dependency Injection

- Service Locator
- Abstract Factory
- Factory Method

Patrones estructurales:

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

Patrones de comportamiento:

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

Links

Links