Son unos principios que deberíamos tener como base antes de proponer una arquitectura de software para el desarrollo de nuestras aplicaciones.

Estos principios nos brindan formas de pasar del código estrechamente acoplado y pobremente encapsulado a los resultados deseados de las necesidades reales de una empresa.

Siguiendo SOLID se desarrolla un código que pueda ser **escalable a un futuro**

Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la **alta cohesión** y el **bajo acoplamiento**.

**SOLID**

SOLID es un acrónimo de lo siguiente.

* **S**: Single Responsibility Principle (SRP)
* **O**: Open/Closed Principle (OCP)
* **L**: Liskov substitution Principle (LSP)
* **I**: Interface Segregation Principle (ISP)
* **D**: Dependency Inversion Principle (DIP)

**Conclusión**

SOLID es lo que deberíamos intentar aplicar en nuestros proyectos para hacer código de calidad

* S: Single Responsibility Principle (SRP)

“Cada módulo de software debe tener una sola razón para cambiar”

* **O**: Open/Closed Principle (OCP)



“Si queremos añadir nuevo código, lo ideal sería poder construir sobre lo que ya existe, sin tener que hacer modificaciones grandes

El principio abierto / cerrado dice "Un módulo / clase de software está abierto para extensión y cerrado para modificación".

En pocas palabras: **si queremos añadir nuevo código, lo ideal sería poder construir sobre lo que ya existe, sin tener que hacer modificaciones grandes**.

Supongamos que tenemos una clase de  
rectángulo con las propiedades Altura y Ancho.

public class Rectangle{

public double Height {get;set;}

public double Wight {get;set; }

}

Nuestra aplicación necesita la capacidad de calcular el área total de una colección de rectángulos.

Nos creamos otra clase para el cálculo del área.

public class AreaCalculator

{

public double TotalArea(Rectangle[] arrRectangles)

{

double area;

foreach(var objRectangle in arrRectangles)

{

area += objRectangle.Height \* objRectangle.Width;

}

return area;

}

}

¿Podemos ampliar nuestra aplicación para que pueda calcular el área no solo de los rectángulos, sino también el área de los círculos?

Podemos cambiar un poco el método TotalArea, de modo que pueda aceptar una matriz de objetos como un argumento.

Verificamos el tipo de objeto en el bucle y hacemos el cálculo del área en función del tipo de objeto.

public class Rectangle

{

public double Height {get;set;}

public double Wight {get;set; }

}

public class Circle

{

public double Radius {get;set;}

}

public class AreaCalculator

{

public double TotalArea(object[] arrObjects)

{

double area = 0;

Rectangle objRectangle;

Circle objCircle;

foreach(var obj in arrObjects)

{

if(obj is Rectangle)

{

area += obj.Height \* obj.Width;

}

else

{

objCircle = (Circle)obj;

area += objCircle.Radius \* objCircle.Radius \* Math.PI;

}

}

return area;

}

}

Hemos terminado con el cambio. Aquí hemos introducido con éxito Circle en nuestra aplicación.

Podemos agregar un Triángulo y calcular su área agregando un bloque más "if" en el método TotalArea de AreaCalculator.

Pero cada vez que introducimos una nueva forma necesitamos alterar el método TotalArea.

Así que la clase AreaCalculator no está cerrada por modificación.

¿Cómo podemos hacer nuestro diseño para evitar esta situación?

Generalmente, podemos hacer esto refiriéndonos a las abstracciones para dependencias, como interfaces o clases abstractas, en lugar de usar clases concretas.

public abstract class Shape

{

public abstract double Area();

}

Heredando de Shape, las clases Rectangle y Circle ahora se ven así:

public class Rectangle: Shape

{

public double Height {get;set;}

public double Width {get;set;}

public override double Area()

{

return Height \* Width;

}

}

public class Circle: Shape

{

public double Radius {get;set;}

public override double Area()

{

return Radius \* Radus \* Math.PI;

}

}

Cada forma contiene su área con su propia forma de funcionalidad de cálculo y nuestra clase AreaCalculator será más simple que antes.

public class AreaCalculator

{

public double TotalArea(Shape[] arrShapes)

{

double area=0;

foreach(var objShape in arrShapes)

{

area += objShape.Area();

}

return area;

}

}

Ahora nuestro código sigue a SRP y OCP.

Siempre que introduzca una nueva forma derivando de la clase abstracta "Shape", no necesita cambiar la clase "AreaCalculator".

* **L**: Liskov substitution Principle (LSP)



“Se debería poder utilizar cualquier clase derivada en lugar de una clase principal y hacer que se comporte de la misma manera sin modificaciones".

Este principio es el que más me costo entender sobre todo por su compleja definición.

Esto es lo que nos dice la Wikipedia. Si **S** es un subtipo de **T** , entonces los objetos de tipo **T** en un programa de computadora pueden ser sustituidos por objetos de tipo **S** (es decir, los objetos de tipo **S** pueden sustituir objetos de tipo **T** ), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.) -

¿Os ha quedado claro?,

Más sencillo.

El principio de sustitución de Liskov (LSP) establece que "debería poder utilizar cualquier clase derivada en lugar de una clase principal y hacer que se comporte de la misma manera sin modificaciones".

Asegura que una clase derivada no afecte el comportamiento de la clase principal, en otras palabras, que una clase derivada debe ser sustituible por su clase base.

Ahora vamos a ver un ejemplo para aprender cómo un diseño puede violar el LSP. Supongamos que necesitamos crear una aplicación para administrar datos usando un grupo de texto de archivos SQL.

Aquí necesitamos escribir la funcionalidad para cargar y guardar el texto de un grupo de archivos SQL en el directorio de la aplicación.

Entonces, necesitamos una clase que maneje la carga y guarde el texto de un grupo de archivos SQL junto con la Clase SqlFile.

public class SqlFile

{

public string FilePath {get;set;}

public string FileText {get;set;}

public string LoadText()

{

*/\* Code to read text from sql file \*/*

}

public string SaveText()

{

*/\* Code to save text into sql file \*/*

}

}

public class SqlFileManager

{

public List<SqlFile> lstSqlFiles {get;set}

public string GetTextFromFiles()

{

StringBuilder objStrBuilder = new StringBuilder();

foreach(var objFile in lstSqlFiles)

{

objStrBuilder.Append(objFile.LoadText());

}

return objStrBuilder.ToString();

}

public void SaveTextIntoFiles()

{

foreach(var objFile in lstSqlFiles)

{

objFile.SaveText();

}

}

}

Hemos terminado con nuestra parte.

Después de un tiempo, nuestro cliente potencial puede decirnos que es posible que tengamos algunos archivos de solo lectura en la carpeta de la aplicación, por lo que debemos restringir el flujo cada vez que intente guardarlos.

Podemos hacerlo creando una clase "ReadOnlySqlFile" que hereda la clase "SqlFile" y necesitamos modificar el método SaveTextIntoFiles () introduciendo una condición para evitar llamar al método SaveText () en las instancias ReadOnlySqlFile.

public class SqlFile

{

public string LoadText()

{

*/\* Code to read text from sql file \*/*

}

public void SaveText()

{

*/\* Code to save text into sql file \*/*

}

}

public class ReadOnlySqlFile: SqlFile

{

public string FilePath {get;set;}

public string FileText {get;set;}

public string LoadText()

{

*/\* Code to read text from sql file \*/*

}

public void SaveText()

{

*/\* Throw an exception when app flow tries to do save. \*/*

throw new IOException("Can't Save");

}

}

Para evitar una excepción, necesitamos modificar "SqlFileManager" agregando una condición al bucle.

public class SqlFileManager

{

public List<SqlFile> lstSqlFiles {get;set}

public string GetTextFromFiles()

{

StringBuilder objStrBuilder = new StringBuilder();

foreach(var objFile in lstSqlFiles)

{

objStrBuilder.Append(objFile.LoadText());

}

return objStrBuilder.ToString();

}

public void SaveTextIntoFiles()

{

foreach(var objFile in lstSqlFiles)

{

*//Check whether the current file object is read only or not.If yes, skip calling it's*

*// SaveText() method to skip the exception.*

if(! objFile is ReadOnlySqlFile)

objFile.SaveText();

}

}

}

No podemos usar esta clase ReadOnlySqlFile como sustituto de su padre sin alterar el código de SqlFileManager.

Así que podemos decir que este diseño no está siguiendo LSP.

Hagamos este diseño siguiendo el LSP.

Aquí introduciremos interfaces para que la clase SqlFileManager sea independiente del resto de los bloques.

public interface IReadableSqlFile

{

string LoadText();

}

public interface IWritableSqlFile

{

void SaveText();

}

Ahora implementamos IReadableSqlFile a través de la clase ReadOnlySqlFile que solo lee el texto de los archivos de solo lectura.

public class ReadOnlySqlFile: IReadableSqlFile

{

public string FilePath {get;set;}

public string FileText {get;set;}

public string LoadText()

{

*/\* Code to read text from sql file \*/*

}

}

Aquí implementamos tanto IWritableSqlFile como IReadableSqlFile en una clase SqlFile mediante la cual podemos leer y escribir archivos.

public class SqlFile: IWritableSqlFile,IReadableSqlFile

{

public string FilePath {get;set;}

public string FileText {get;set;}

public string LoadText()

{

/\* Code to read text from sql file \*/

}

public void SaveText()

{

/\* Code to save text into sql file \*/

}

}

Ahora el diseño de la clase SqlFileManager se vuelve así:

public class SqlFileManager

{

public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)

{

StringBuilder objStrBuilder = new StringBuilder();

foreach(var objFile in aLstReadableFiles)

{

objStrBuilder.Append(objFile.LoadText());

}

return objStrBuilder.ToString();

}

public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)

{

foreach(var objFile in aLstWritableFiles)

{

objFile.SaveText();

}

}

}

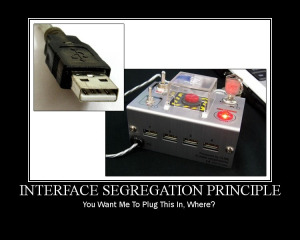
Aquí, el método GetTextFromFiles () obtiene solo la lista de instancias de clases que implementan la interfaz IReadOnlySqlFile.

Eso significa que las instancias de clase SqlFile y ReadOnlySqlFile. Y el método SaveTextIntoFiles () obtiene solo las instancias de la lista de la clase que implementa la interfaz IWritableSqlFiles, en otras palabras, las instancias de SqlFile en este caso.

Ahora podemos decir que nuestro diseño está siguiendo el LSP.

Tener a la mano patrones de diseño relacionados con la herencia pueden ser muy útiles. Algunos ejemplos:

* Método plantilla.
* Composite.
* Estrategia.
* Estado.
* **I**: Interface Segregation Principle (ISP)



Los clientes no deben ser forzados a implementar interfaces que no usan. En lugar de una sola interfaz, se prefieren muchas interfaces pequeñas basadas en grupos de métodos, cada uno de los cuales sirve a un submódulo". Debemos tener cuidado con definir interfaces con muchos métodos. es mejor tener interfaces pequeñas, con pocos métodos muy relacionados (alta cohesión), en lugar de tener interfaces voluminosas que obligan a definir muchos métodos a quien las implementa.

El Principio de Segregación de Interfaz establece que "los clientes no deben ser forzados a implementar interfaces que no usan. En lugar de una sola interfaz, se prefieren muchas interfaces pequeñas basadas en grupos de métodos, cada uno de los cuales sirve a un submódulo".

Podemos definirlo de otra manera.

Una interfaz debe estar más relacionada con el código que la usa que con el código que la implementa.

Vamos a empezar con un ejemplo que rompe ISP.

Supongamos que necesitamos construir un sistema para una empresa de IT que tenga roles como TeamLead y Programmer, donde TeamLead divide una tarea enorme en tareas más pequeñas y las asigna a sus programadores o puede trabajar directamente en ellas.

Según las especificaciones, necesitamos crear una interfaz y una clase TeamLead para implementarla.

public Interface ILead

{

void CreateSubTask();

void AssginTask();

void WorkOnTask();

}

public class TeamLead : ILead

{

public void AssignTask()

{

*//Code to assign a task.*

}

public void CreateSubTask()

{

*//Code to create a sub task*

}

public void WorkOnTask()

{

*//Code to implement perform assigned task.*

}

}

El diseño se ve bien por ahora.

Posteriormente, se introduce en el sistema otra función como Administrador, que asigna tareas a TeamLead y no trabajará en las tareas.

public class Manager: ILead

{

public void AssignTask()

{

*//Code to assign a task.*

}

public void CreateSubTask()

{

*//Code to create a sub task.*

}

public void WorkOnTask()

{

throw new Exception("Manager can't work on Task");

}

}

Ya que el Administrador no puede trabajar en una tarea y al mismo tiempo nadie puede asignar tareas al Administrador, esta WorkOnTask () no debe estar en la clase Administrador.

Pero estamos implementando esta clase desde la interfaz de ILead, necesitamos proporcionar un método concreto.

Aquí forzamos a la clase Manager a implementar un método WorkOnTask () sin un propósito. Esto está mal. El diseño viola el ISP.

Corrigamos el diseño.

Una interfaz que proporciona un contrato para WorkOnTask ().

public interface IProgrammer

{

void WorkOnTask();

}

Una interfaz que proporciona contratos para gestionar las tareas:

public interface ILead

{

void AssignTask();

void CreateSubTask();

}

Entonces la implementación se convierte en:

public class Programmer: IProgrammer

{

public void WorkOnTask()

{

*//code to implement to work on the Task.*

}

}

public class Manager: ILead

{

public void AssignTask()

{

*//Code to assign a Task*

}

public void CreateSubTask()

{

*//Code to create a sub taks from a task.*

}

}

TeamLead puede administrar tareas y puede trabajar en ellas si es necesario.

La clase TeamLead debe implementar las interfaces de IProgrammer e ILead.

public class TeamLead: IProgrammer, ILead

{

public void AssignTask()

{

*//Code to assign a Task*

}

public void CreateSubTask()

{

*//Code to create a sub task from a task.*

}

public void WorkOnTask()

{

*//code to implement to work on the Task.*

}

}

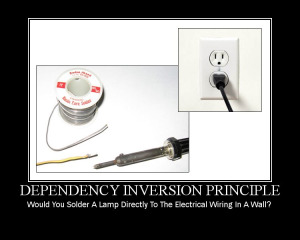
Aquí separamos responsabilidades / propósitos, los distribuimos en múltiples interfaces y también proporcionamos un buen nivel de abstracción

Como conclusión

Debemos tener cuidado con definir interfaces con muchos métodos. De acuerdo a este principio, **es mejor tener interfaces pequeñas, con pocos métodos muy relacionados (alta cohesión), en lugar de tener interfaces voluminosas que obligan a definir muchos métodos a quien las implementa**.

Algunos patrones, especialmente los de comportamiento, pueden ser muy convenientes:

* Cadena de responsabilidad.
* **D**: Dependency Inversion Principle (DIP)



Este principio tiene dos ideas importantes de fondo.

1. Los módulos de alto nivel no deberían depender de módulos de bajo nivel, sino de abstracciones.

2. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

**Principios de diseño de software**

Esta parte del curso pretende profundizar en el significado de todas las frases que utilizamos hoy en día y seguramente hayas escuchado, como son KISS, DRY, YAGNI…

### ****You Aren't Gonna Need It****

Literalmente traducido por «No vas a necesitarlo». Este principio esta detrás de otro mucho más largo y feo como es DTSTTCPW utilizado en extreme programming que viene a decir «haz la cosa más simple que pueda funcionar» (Do The Simplest Thing That Could Possibly Work).

Está pensado para utilizarse en la metodología de trabajo de extreme programming junto con las pruebas y la refactorización

### ****¿Qué es KISS?****

Es un acrónimo de la frase inglesa "Keep it simple, Stupid!"

Este patrón lo que nos dice es que cualquier sistema va a funcionar mejor si se mantiene sencillo que si se vuelve complejo. Es decir que la sencillez tiene que ser una meta en el desarrollo y que la complejidad innecesaria debe ser eliminada.

### ****Don't Repeat Yourself (DRY) / Duplication Is Evil (DIE)****

Agrupo estos dos acrónimos porque su significado es el mismo, «no te repitas» y la «duplicación es el mal».

Si ya tienes una funcionalidad existente, es mejor reutilizarla que duplicarla

## **Ley de Demeter: "Habla solo con tus amigos cercanos. No hables con extraños."**

Allá por 1980, un grupo de programadores que trabajaban en un proyecto llamado el sistema Demeter, se dieron cuenta de que ciertos aspectos de su código orientado a objetos hacían que fuera más fácil de mantener y cambiar. Por ejemplo un bajo acoplamiento entre objetos, ocultar la información, entre otras cosas.

Por ello quisieron agrupar todas estas características que veían que mejoraban la mantenibilidad del software en una heurística, en una regla, que los programadores pudieran aplicar en su código para mejorarlo. Así llegaron a la Ley de Demeter.

## **La problemática**

El típico código que huele bastante mal es algo tal que así:

getX().getY().getZ().doSomething()

## **¿Qué es la Ley de Demeter?**

La ley de Demeter básicamente es un mecanismo de detección de acoplamiento, y nos viene a decir que **nuestro objeto no debería conocer las entrañas de otros objetos con los que interactúa**.

Una forma fácil de detectar que se está violando esta Ley es **encontrarte con muchas llamadas concatenadas**. Pero no es una razón 100% fiable ni es el único motivo.

El código anterior se podría escribir de la siguiente forma, y seguiría siendo igual de inválido:

val x = getX()

val y = x.getY()

val z = y.getZ()

z.doSomething()

## **¿Cómo soluciono las violaciones de la Ley de Demeter?**

### ****1. Añadir métodos extra****

getX().doSomething()

Este a su vez dentro llamaría a:

getY().doSomething()

Lo que está haciendo es esconder el problema, no solucionarlo.

### ****2. Arquitectura****

Una buena arquitectura juega un papel muy importante en el desacoplamiento de los distintos módulos del software, y por tanto reducirá bastante la posibilidad de violar esta ley.

### ****3. Comprender mejor tu dominio****

La idea del dominio y en particular de **Domain Driven Design** viene a decirnos que nuestro problema se puede sustentar en unos conceptos clave que lo definen, y teniéndolos muy claros todo el proceso de desarrollo se simplifica mucho.

### ¿Cómo evitamos incumplir la Ley de Demeter?

Si detectamos que estamos incumpliendo la Ley de Demeter, lo primero es aceptar que tenemos un problema de arquitectura en la base de nuestro proyecto y esto puede implicar cambios bastante grandes.

## **Beneficios de cumplir la Ley de Demeter**

– Se reducen las dependencias entre clases y el acoplamiento.

– Se vuelve más sencillo reutilizar las clases.

– El código es más fácil de probar.

– El código es más mantenible, más flexible a los cambios.

### ****Conclusión****

Si detectamos el problema, es importante tratar de solucionarlo con buenas prácticas y no enmascarar un problema que puede comprometer nuestro sistema con el tiempo

El método MoSCoW es una técnica de priorización de requisitos basada en el hecho de que aunque todos los requisitos se consideren importantes es fundamental destacar aquellos que permiten darle un mayor valor al sistema, lo que permite enfocar los trabajos de manera más eficiente.

**M (Must)**: Requisito que tiene que estar implementado en la versión final del producto para que la misma pueda ser considerada un éxito.

**S (Should)**: Requisito de alta prioridad que en la medida de lo posible debería ser incluido en la solución final, pero que llegado el momento y si fuera necesario, podría ser prescindible si hubiera alguna causa que lo justificara.

**C (Could)**: Requisito deseable pero no necesario, se implementaría si hubiera posibilidades presupuestarias y temporales.

**W (Won't)**: Hace referencia a requisitos que están descartados de momento pero que en un futuro podrían ser tenidos de nuevo en cuenta y ser reclasificados en una de las categorías anteriores.

Esta clasificación puede ser modificada durante el proceso de desarrollo y definirse, en el caso de desarrollos iterativos incrementales, prioridades a nivel de iteración.

La técnica de priorización de requisitos MoSCoW, **ayuda a todo el equipo de desarrollo** a entender las consecuencias reales de la asignación de la criticidad

**Patrones de Diseño**

Un patrón de diseño es una forma reutilizable de resolver un problema común.

El concepto de patrón de diseño lleva existiendo desde finales de los 70, pero su verdadera popularización surgió en los 90 con el lanzamiento del libro de Design Pattern de la Banda de los Cuatro (Gang of Four). En él explican 23 patrones de diseño

## **¿Por qué son útiles los patrones de diseño?**

### ****1. Ahorro de tiempo****

### ****2. Validez de tu código****

### ****3. Uso de un lenguaje común****

## **¿Cómo identificar qué patrón encaja con tu problema?**

Este es el punto más complicado **se aprende practicando**.

## **¿Por qué usar patrones de diseño?**

**Los patrones de diseño nos ayudan a cumplir muchos de estos principios o reglas de diseño**. Programación SOLID, control de cohesión y acoplamiento o reutilización de código son algunos de los beneficios que podemos conseguir al utilizar patrones.

## **¿Cuántos patrones de diseño existen?**

Patrones de diseño hay muchos. Muchísimos. Y siguen apareciendo patrones nuevos cada poco tiempo. En este curso veremos los 23 patrones del libro Gang of Four

## **Tipos de Patrones en GoF**

**Patrones Creacionales**

**Patrones Estructurales**

**Patrones de Comportamiento**

Los patrones de diseño pretenden:

* Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
* Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
* Formalizar un vocabulario común entre diseñadores.
* Estandarizar el modo en que se realiza el diseño.
* Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

* Imponer ciertas alternativas de diseño frente a otras.
* Eliminar la creatividad inherente al proceso de diseño.

**Patrones Creacionales**

**Abstract Factory**

**Definición**

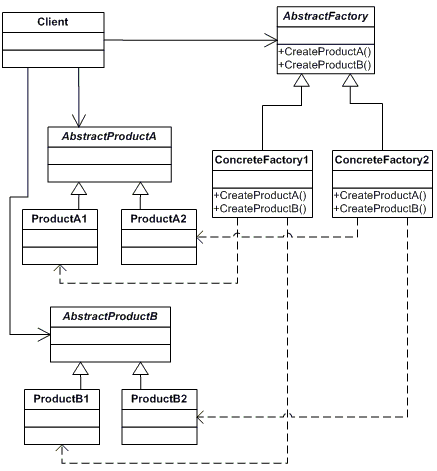
Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

Frecuencia de uso:



Alta

**Diagrama clases UML**



**Participantes**

Las clases que participan en este patrón son:

* **AbstractFactory** **(ContinentFactory)**
  + declara una interfaz para operaciones que crean productos abstractos
* **ConcreteFactory** **(AfricaFactory, AmericaFactory)**
  + implementa las operaciones para crear objetos de productos concretos.
* **AbstractProduct** **(Herbivore, Carnivore)**
  + declara una interfaz para un tipo de objeto de producto
* **Product** **(Wildebeest, Lion, Bison, Wolf)**
  + define un objeto producto a ser creado por la fábrica de concreto correspondiente
  + implementa la interfaz de producto abstracto
* **Client** **(AnimalWorld)**
  + usa interfaces declaradas por las clases AbstractFactory y AbstractProduct

**Código Estructural en C#**

Este código estructural demuestra el patrón de **Abstract Factory** creando jerarquías paralelas de objetos.

Ver Creational Patterns sln

**Salida**

ProductB1 interacts with ProductA1  
ProductB2 interacts with ProductA2

**Código problema Real en C#**

Este código demuestra la creación de diferentes mundos animales para un juego que utiliza diferentes Factorys.

Ver Creational Patterns sln

**Salida**

Lion eats Wildebeest  
Wolf eats Bison

**Patron Builder**

### ****Definicion****

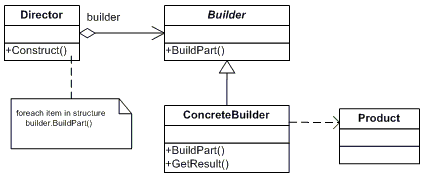
Separa la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones.

Frequencia de uso:



Media baja

### ****Diagrama clases UML****



### ****Participantes****

Las clases y objetos que participan en este patrón son:

* **Builder** **(VehicleBuilder)**
  + especifica una interfaz abstracta para crear partes de un objeto Product
* **ConcreteBuilder** **(MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  + construye y ensambla partes del producto implementando la interfaz Builder
  + define y realiza un seguimiento de la representación que crea
  + proporciona una interfaz para recuperar el producto
* **Director** **(Shop)**
  + construye un objeto usando la interfaz de Builder
* **Product** **(Vehicle)**
  + representa el objeto complejo en construcción. ConcreteBuilder construye la representación interna del producto y define el proceso mediante el cual se ensambla
  + incluye clases que definen las partes constituyentes, incluidas las interfaces para ensamblar las partes en el resultado final

### ****Código Estructural en C#****

Este código estructural muestra el patrón de **Builder** en el que los objetos complejos se crean paso a paso.

Ver Creational Patterns sln

##### **Salida**

Product Parts ———-  
PartA  
PartB

Product Parts ———-  
PartX  
PartY

### ****Código problema Real en C#****

Este código demuestra el patrón de **Builder** en el que los diferentes vehículos se ensamblan paso a paso.

Ver Creational Patterns sln

##### **Salida**

Vehicle Type: Scooter  
Frame : Scooter Frame  
Engine : none

# **Wheels: 2**

# **Doors : 0**

Vehicle Type: Car  
Frame : Car Frame  
Engine : 2500 cc

# **Wheels: 4**

# **Doors : 4**

Vehicle Type: MotorCycle  
Frame : MotorCycle Frame  
Engine : 500 cc

# **Wheels: 2**

# **Doors : 0**

**FACTORTY METHOD**

### ****Definición****

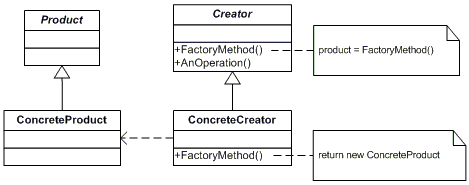
Define una interfaz para crear un objeto, pero deja que las subclases decidan de qué clase crear una instancia. **Factory Method** permite que una clase difiera la creación de instancias a subclases.

Frecuencia de uso:



Alta

### ****Diagrama clases UML****



### ****Participantes****

Las clases y objetos que participan en este patrón son:

* **Product** **(Page)**
  + define la interfaz de los objetos que crea el Factory method
* **ConcreteProduct** **(SkillsPage, EducationPage, ExperiencePage)**
  + implementa la interfaz del Product
* **Creator** **(Document)**
  + declara el Factory method, que devuelve un objeto de tipo Product. El Creator también puede definir una implementación predeterminada del Factory method que devuelve un objeto ConcreteProduct predeterminado..
* **ConcreteCreator** **(Report, Resume)**
  + Hace override del Factory method para devolver una instancia de ConcreteProduct.

### ****Código Estructural en C#****

Este código estructural demuestra el Factory method que ofrece una gran flexibilidad en la creación de diferentes objetos.

Ver Creational Patterns sln

##### **Salida**

Created ConcreteProductA  
Created ConcreteProductB

### ****Código problema Real en C#****

Este código demuestra que el Factory method ofrece flexibilidad en la creación de diferentes documentos.

Ver Creational Patterns sln

##### **Salida**

Resume ———-  
SkillsPage  
EducationPage  
ExperiencePage

Report ———-  
IntroductionPage  
ResultsPage  
ConclusionPage  
SummaryPage  
BibliographyPage

**Prototype**

### ****Definición****

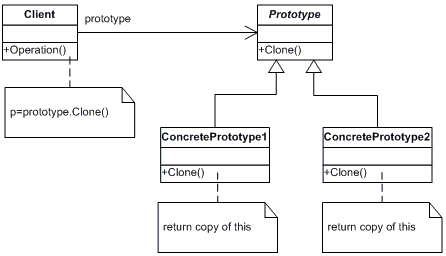
Especifica el tipo de objetos para crear utilizando una instancia prototípica y cree nuevos objetos copiando este prototipo..

Frequencia de uso:



Media

### ****Diagrama clases UML****



### ****Participantes****

Las clases y objetos que participan en este patrón son:

* **Prototype** **(ColorPrototype)**
  + declara una interfaz para clonación propia
* **ConcretePrototype** **(Color)**
  + implementa una operación de clonación propia.
* **Client** **(ColorManager)**
  + crea un nuevo objeto pidiendo a un prototipo que se clone a sí mismo

### ****Código Estructural en C#****

Este código estructural muestra el patrón de **Prototype** en el que se crean nuevos objetos al copiar objetos preexistentes (prototipos) de la misma clase.

Ver Creational Patterns sln

##### **Output**

Cloned: I  
Cloned: II

### ****Código problema Real en C#****

Este código del mundo real muestra el patrón **Prototype** en el que se crean nuevos objetos de color al copiar colores preexistentes y definidos por el usuario del mismo tipo.

Ver Creational Patterns sln

##### **Output**

Cloning color RGB: 255, 0, 0  
Cloning color RGB: 128,211,128  
Cloning color RGB: 211, 34, 20

**Singleton**

### ****Definición****

Se asegura de que una clase tenga solo una instancia y proporciona un punto global de acceso a ella..

Frecuencia de uso:



Media alta

### ****Diagrama clases UML****

### 

### 

### 

### ****Participantes****

Las clases y objetos que participan en este patrón son:

* **Singleton** **(LoadBalancer)**
  + Define una operación de instancia que permite a los clientes acceder a su instancia única. La instancia es una operación de clase.
  + Responsable de crear y mantener su propia instancia única.

### ****Código Estructural en C#****

Este código estructural demuestra el patrón de **Singleton** que asegura que solo se pueda crear una única instancia (el singleton) de la clase.

Ver Creational Patterns sln

##### **Salida**

Objects are the same instance

### ****Código problema Real en C#****

Este código muestra el patrón **Singleton** como un objeto LoadBalancing

Ver Creational Patterns sln

##### **Output**

Same instance

ServerIII  
ServerII  
ServerI  
ServerII  
ServerI  
ServerIII  
ServerI  
ServerIII  
ServerIV  
ServerII  
ServerII  
ServerIII  
ServerIV  
ServerII  
ServerIV