ESTRUCTURA

1.- Fundamentos Teóricos de OOP.  
   - Definir que es el: Polimorfismo, herencia, encapsulación.  
   - Que es una clase abstracta interfaz y para q se usan.  
   - Que es una clase y u objeto.

- Clases Internas y Locales.  
  
2.- Preguntas Técnicas del Lenguaje en cuestión (C++, Java).  
   - como se declaran constantes.  
   - como se utilizan las hebras, excepciones, etc...  
   - como definir una clase abstracta, método abstracto, interfaz, método abstracto.  
   - q son las clases, atributos o métodos finales y  como se definen.

- Clases Internas.

- Tipos de Casting en C++.  
  
3.- Bases de Datos. (Allí usan MySQL con paquete XSQL).  
   - como harías un trigger o un procedimiento PL/SQL.

- Conceptos: Primary Key, Foreign Key, Índices, Problemas asociados al definir muchos índices, ….  
   - como detectas que una consulta va lenta y q hacer para optimizarla.

BLOQUE 1: POO

1. **Definición**:

La POO es una metodología de programación que consiste en la abstracción de un problema en base a un conjunto de objetos que interactúan entre sí. Los pilares básicos de la POO son: Abstracción, Herencia, Polimorfismo y la Encapsulación.

1. **Abstracción**:

Capacidad de definir algo del mundo real a través de objetos que mantienen lo esencial para el problema en cuestión y lo que los diferencian unos de otros.

1. **Herencia**:

Establece una jerarquía de clases: SuperClase – SubClase.

La herencia permite que se puedan definir nuevas clases basadas en clases existentes, lo cual facilita re-utilizar código previamente desarrollado. Si una clase deriva de otra hereda todas sus variables y métodos. La clase derivada puede ***añadir*** nuevas variables y métodos y/o ***redefinir*** las variables y métodos heredados.

1. **Polimorfismo**:

Los objetos de distintas clases pertenecientes a una misma jerarquía o que

implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo.

1. **Encapsulación**:

La encapsulación es un mecanismo que consiste en organizar datos y métodos de un objeto, evitando el acceso a datos por cualquier otro medio distinto a los especificados. Por lo tanto, la encapsulación garantiza la integridad de los datos que contiene un objeto.

Esta ligada a la ocultación y por tanto a la restricción del acceso.

Las clases pueden ser declaradas como públicas (***public***) y como ***package***

(accesibles sólo para otras clases del ***package***). Las variables miembro y los métodos pueden ser ***public***, ***private***, ***protected*** y ***package***. De esta forma se puede controlar el acceso y evitar un uso inadecuado.

1. **Sobrecarga**:

Consiste en definir métodos con el mismo nombre pero con parámetros diferentes, ya sea en número o tipos. Es una alternativa al polimorfismo estático o por vinculación temprana.

1. **Objeto**:

Es una abstracción de algo que tiene propia identidad en el dominio de un problema a solucionar con POO y que se compone de:

Atributos: datos asociados, características o propiedades. Sus valores concretos determinan el estado. Pueden ser propios de cada objeto o comunes a todos. También se pueden llamar variables.

Métodos: comportamiento que tiene el objeto relacionado con la forma de tratar sus atributos, que a su vez define la interface a la hora de interactuar con el mismo. Estos comportamientos pueden ser propios de cada objeto o comunes al mismo “Tipo” de objetos.

Un objeto es una instancia de una clase.

1. **Clases**:

Las clases son la definición formal de los objetos , es decir, el patrón con el que se crean.

1. **Clase Abstracta**:

Es una clase que se utiliza únicamente para la definición de otras en la jerarquía según la herencia. No permiten que se instancien objetos.

1. **Interfaz**:

La interfaz de un objeto son el conjunto de métodos con los que se puede interactuar con dicho objeto.

Con respecto a los lenguajes de POO representa la definición de un conjunto de métodos que se deben implementar en la clase que lo implemente. En c++ y java se tratan de forma diferente, no existiendo propiamente el concepto en c++ debiéndose utilizar clases abstractas.

Son útiles en términos de polimorfismo y además en java son una alternativa a la herencia múltiple implementada de serie en c++.

BLOQUE 2: DETALLES TÉCNICOS DE JAVA Y C++

La principal diferencia entre ambos lenguajes es que en la práctica JAVA define cada clase en un fichero “.java” en el nombre de la clase pública debe coincidir con el del respectivo fichero. En C++ no existe esta restricción y permite la creación de las clases que se quieran en un mismo fichero, pero desdobla la definición e implementación en dos ficheros: “.hpp” o “.h” y “.cpp”.

1. Clases:

|  |  |
| --- | --- |
| C++ | JAVA |
| **.h:**  **Namespace NombreNameEspace**  **{**  **class** <identificador de clase> [<:lista de clases base>] {  <lista de miembros>  } [<lista de identificadores de objetos>];  }  Las clase por defecto son públicas. | package nombrePaq;  import java.lang.\*;  [public] **class** Classname extends SuperClase implements Interface  {  // definición de variables y métodos  ...  }  Nota: Las clases pueden ser públicas o de paquete (accesible por clases del mismo package, opción por defecto) |

1. Interfaces:

|  |  |
| --- | --- |
| C++ | JAVA |
| **.h:**  **class** <identificador de clase> [<:lista de clases base>] {  **//TODAS LOS MÉTODOS SON VIRTUALES PUROS =0.**  }; | import java.awt.Graphics;  public interface Dibujable  {  public void setPosicion(double x, double y);  public void dibujar(Graphics dw);  } |

1. Clases Abstractas:

|  |  |
| --- | --- |
| C++ | JAVA |
| **.h:**  **class** <identificador de clase> [<:lista de clases base>] {  **//COMO MÍNIMO UN MÉTODO VIRTUAL PURO.**  }; | public abstract class NombreClass  { ... }  NOTA:  - Puede tener unos o todos lo métodos abstractos y no se definen.  - Se definen con “abstract”.  - Métodos static no pueden ser abst. |

1. Clases/Atributos/Métodos Finales:

|  |  |
| --- | --- |
| C++ | JAVA |
| **Atributo Final o Constante:** Atributo que no puede cambiar. | |
| Palabra reservada “const”. | Palabra reservada “final”. Si no se inicializa a partir de la primera vez que se instancia adquiere un valor constante. |
| **Clase Final: Clase que no puede ser clase derivada o superclase de otra.** | |
| **NO EXISTE DIRECTAMENTE** | public final class NombreClass  { ... } |
| **Método Final: Método que no se puede redefinir.** | |
| **NO EXISTE** | public final void nombreMetodo()  { ... } |

1. Excepciones:

Una excepción es un error o malfuncionamiento en tiempo de ejecución. El tratamiento o manejo de excepciones consiste en controlar estas situaciones e intentar tomar una determinación al respecto.

|  |  |
| --- | --- |
| C++ | JAVA |
| **try, catch y throw.**  Tipos primitivos o clases. Normalmente se derivan de la clase exception.  En la definición del método se puede indicar las excepciones que se disparan.  class exception {  public:  exception() throw() { }  virtual ~exception() throw();  virtual const char\* what() const throw();  };  Excepciones predefinidas heredan de exception:  std::bad\_alloc // Al operador new std::bad\_cast // Al operador dynamic\_cast<> std::bad\_typeid // Al operador typeid std::bad\_exception // Cuando se viola una especificación  Captura genérica catch(…)  No existe jerarquía en las excepciones como en java, al lanzarse se busca de dentro a fuera el catch que captura el tipo en concreto.  Ejemplo:  try {  x = new int[y];  delete[] x;  }  catch(std::bad\_alloc&) {  *cout* << "Memoria insuficiente" << *endl*;  } | ***try***,  ***catch***, ***throw***, ***throws*** y ***finally***  En JAVA las excepciones se establecen con clases que derivan de Exception que a su vez deriva de Throwable dentro del paquete java.lang.  Tipos de excepciones:   * **Implícitas**: a las que no se obligan a capturar y que se gestionan con RuntimeException. Pe: java.lang.NullpointerException. * **Explícitas**: el resto, que si obliga a gestionar. |

1. Hebras:

|  |  |
| --- | --- |
| C++ | JAVA |
| POSIX: pthread  WINDOWS: ws2\_32.dll |  |

1. Sockets:

C++. SERVER.

/\* Estos son los ficheros de cabecera usuales \*/

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#define PORT 3550 /\* El puerto que será abierto \*/

#define BACKLOG 2 /\* El número de conexiones permitidas \*/

main()

{

int fd, fd2; /\* los ficheros descriptores \*/

struct sockaddr\_in server;

/\* para la información de la dirección del servidor \*/

struct sockaddr\_in client;

/\* para la información de la dirección del cliente \*/

int sin\_size;

/\* A continuación la llamada a socket() \*/

if ((fd=socket(AF\_INET, SOCK\_STREAM, 0)) == -1 ) {

printf("error en socket()\n");

exit(-1);

}

server.sin\_family = AF\_INET;

server.sin\_port = htons(PORT);

/\* ¿Recuerdas a htons() de la sección "Conversiones"? =) \*/

server.sin\_addr.s\_addr = INADDR\_ANY;

/\* INADDR\_ANY coloca nuestra dirección IP automáticamente \*/

bzero(&(server.sin\_zero),8);

/\* escribimos ceros en el reto de la estructura \*/

/\* A continuación la llamada a bind() \*/

if(bind(fd,(struct sockaddr\*)&server,

sizeof(struct sockaddr))==-1) {

printf("error en bind() \n");

exit(-1);

}

if(listen(fd,BACKLOG) == -1) { /\* llamada a listen() \*/

printf("error en listen()\n");

exit(-1);

}

while(1) {

sin\_size=sizeof(struct sockaddr\_in);

/\* A continuación la llamada a accept() \*/

if ((fd2 = accept(fd,(struct sockaddr \*)&client,

&sin\_size))==-1) {

printf("error en accept()\n");

exit(-1);

}

printf("Se obtuvo una conexión desde %s\n",

inet\_ntoa(client.sin\_addr) );

/\* que mostrará la IP del cliente \*/

send(fd2,"Bienvenido a mi servidor.\n",22,0);

/\* que enviará el mensaje de bienvenida al cliente \*/

close(fd2); /\* cierra fd2 \*/

}

}

BLOQUE 4: ORM SQL

Glosario

¿Qué es un ORM?

Object-Relational mapping, o lo que es lo mismo, **mapeo de objeto-relacional**, es un modelo de programación que consiste en la transformación de las tablas de una base de datos, en una serie de entidades que simplifiquen las tareas básicas de acceso a los datos para el programador.

Desde hace muchos años el lenguaje más usado para acceder a las bases de datos relacionales ha sido el SQL. ¿Por qué entonces cambiar y pasarse a un ORM?

**¿Por qué usar un ORM?**

Aunque el lenguaje SQL se usa para acceder a muchas de las bases de datos existentes, existen múltiples varianzas en las funciones que los distintos SGBD han usado. Un ejemplo muy sencillo sería delimitar el número de registros de una consulta:

SELECT TOP 10 \* FROM usuarios //SqlServer

SELECT \* FROM usuarios LIMIT 10 //MySQL

SELECT \* FROM usuarios WHERE rownum<=20; //Oracle

Tres de las bases de datos más importantes, y como veis, para algo tan fácil vemos diferencias. Esto para el programador supone tener que conocer el lenguaje para cada Base de datos, y más importante aún, si en un futuro se desea migrar la aplicación, habría que reescribir gran número de las consultas.

Esto el ORM al tener un capa intermedia, **abstrae al programador de la base de datos y le centra en el desarrollo de la aplicación**.

Otro punto importante es la **facilidad de trabajo**, un ORM, nos facilita las labores básicas de cualquier acceso a datos , el CRUD (Create, Read, Update y Delete). Realizando todas estas labores a través de un lenguaje de alto nivel orientado a objetos. Ahora que ya sabemos **que es un ORM** y **porque usarlo**, vamos a ver sus ventajas y desventajas.

**Ventajas y desventajas de un ORM**

* Ventajas
  + Facilidad y velocidad de uso
  + Abstracción de la base de datos usada.
  + Seguridad de la capa de acceso a datos contra ataques.
* Desventajas
  + En entornos con gran carga poner una capa más en el proceso puede mermar el rendimiento.
  + Aprender el nuevo lenguaje del ORM.

**ORMs más utilizados**

Casi todos los lenguajes de alto nivel actualmente disponen de alguna solución de este tipo, una de las más conocidas es Hibernate para JAVA, pero existen muchas más:

* Java => Hibernate, iBatis, Ebean, etc..
* .NET=> Entity Framework, nHibernate, etc..
* PHP=> Doctrine, Propel, ROcks, Torpor, etc..

BLOQUE 4: C# .NET

### 2. What are strong references and weak references in GC?

The garbage collector cannot collect an object in use by an application while the application’s code can reach that object. The application is said to have a strong reference to the object.

A weak reference permits the garbage collector to collect the object while still allowing the application to access the object. A weak reference is valid only during the indeterminate amount of time until the object is collected when no strong references exist. When you use a weak reference, the application can still obtain a strong reference to the object, which prevents it from being collected. However, there is always the risk that the garbage collector will get to the object first before a strong reference is re-established.

Weak references are useful for objects that use a lot of memory, but can be recreated easily if they are reclaimed by garbage collection.

### 5. What is the difference between Var and Dynamic in C#?

|  |  |
| --- | --- |
| **var** | **dynamic** |
| Introduced in C# 3.0 | Introduced in C# 4.0 |
| Statically typed – This means the type of variable declared is decided by the compiler at compile time. | Dynamically typed – This means the type of variable declared is decided by the compiler at run time. |
| var type of variables are required to be initialized at the time of declaration or else they encounter the compile time error: Implicitly-typed local variables must be initialized. | No need to initialize at the time of declaration. |
| e.g., var str=”I am a string”; | e.g., dynamic str; |
| Looking at the value assigned to the variable str, the compiler will treat the variable str as string. | str=”I am a string”; //Works fine and compiles |
| Errors are caught at compile time. | Errors are caught at runtime |
| Since the compiler knows about the type and the methods and properties of the type at the compile time itself | Since the compiler comes to about the type and the methods and properties of the type at the run time. |
| Intellisense help is available for the var type of variables. This is because, its type is inferred by the compiler from the type of value it is assigned and as a result, the compiler has all the information related to the type | Intellisense help is not available for dynamic type of variables since their type is unknown until run time. So intellisense help is not available. Even if you are informed by the compiler as “This operation will be resolved at run-time”. |
| It will  throw a compilation error since the variable is not initialized. The compiler needs that this variable should be initialized so that it can infer a type from the value. | It Will compile successfully |

### 6. What is the difference between action and func in C#?

Action is a delegate (pointer) to a method, that takes zero, one or more input parameters, but does not return anything.  
   
Func is a delegate (pointer) to a method, that takes zero, one or more input parameters, and returns a value (or reference).

### Example:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class Program  {      static void Main(string[] args)      {          Action<int> myAction = new Action<int>(DoSomething);          myAction(123);           // Prints out "123"                                   // can be also called as myAction.Invoke(123);            Func<int, double> myFunc = new Func<int, double>(CalculateSomething);          Console.WriteLine(myFunc(5));   // Prints out "2.5"      }        static void DoSomething(int i)      {          Console.WriteLine(i);      }        static double CalculateSomething(int i)      {          return (double)i/2;      }  } |

### 11. What is the difference between Boxing and Unboxing?

#### **Boxing:**

* The operation of Converting a Value Type to a Reference Type is called **Boxing**
* Boxing is used to store value types in the garbage-collected heap.
* Boxing is an implicit conversion of a value type to the type object or to any interface type implemented by this value type.
* Boxing a value type allocates an object instance on the heap and copies the value into the new object



|  |  |
| --- | --- |
| 1  2 | int Val = 10;  Object Obj = Val; //Boxing |

The first line we created a Value Type Val and assigned a value to Val.

The second line , we created an instance of Object Obj and assign the value of Val to Obj.

From the above operation (Object Obj = i ) we saw converting a value of a Value Type into a value of a corresponding Reference Type .

These types of operation is called Boxing.

#### **Unboxing:**

* Unboxing is simply the opposite of boxing.
* In it values are again shifted from the heap to the stack.
* In unboxing first of all it checks for the boxed object value and then it goes for replica creation
* It converts an object type back into the value type.
* It is an explicit operation using C-style casting.



|  |  |
| --- | --- |
| 1  2  3  4 | int Val = 1;  Object Obj = Val; //Boxing    int i = (int)Obj; //Unboxing |

The first two line shows how to Box a Value Type .

The next line (int i = (int) Obj) shows extracts the Value Type from the Object .

That is converting a value of a Reference Type into a value of a Value Type. This operation is called UnBoxing.

Boxing and UnBoxing are computationally expensive processes. When a value type is boxed, an entirely new object must be allocated and constructed , also the cast required for UnBoxing is also expensive computationally.

Example:

Let’s look at the reference example.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using System;  using System.Windows.Forms;    namespace WindowsApplication1  {   public partial class Form1 : Form      {          public Form1()          {              InitializeComponent();          }          private void button1\_Click(object sender, EventArgs e)          {              int Val = 10;              Object Obj = Val;       //Boxing              int i = (int)Obj;       //Unboxing              MessageBox.Show("The value is   : " + i);          }      }  } |

So boxing is the process of taking a value type, putting it inside a new object on the heap,and storing a reference to it on the stack. Unboxing is the exact opposite: It takes the item from the heap and returns a value type that contains the value from the heap.

If you execute an invalid unbox operation, the runtime will throw an InvalidCastException.

when boxing and unboxing happen, you need to explicitly cast your object from a reference to a value type.

There are some performance implications with each box and unbox operation.

The boxing and unboxing operations can hurt performance; however, now that you have generic support in the .NET Framework, this is less of an issue because you can store value types in a collection without boxing them.

### 12. What is the difference between Compile time Exception and Runtime Exception?

Compile time is where your compiler transforms your source code to a machine understandable language.

During the compile time, it processes through various stages:  
Creation of Symbol table, Syntax analysis, Semantic analysis, Code optimization, Code Generation & Error Handling.

Runtime is during the execution process(Eg: Page request is made. or looping through a variable instances, etc). Runtime errors are handles after the successful compilation.

Example:  
The static variables are allocated with memory in the compile time.  
The variables that are created at runtime(during the execution process), the memory is allocated for them at run time.

The compile time errors may occur with an error in syntax.

This run time errors may be based on the user input like divide by zero exception, stack over flow, pointer unavailability, wrong address reference, referring null string, etc.

### 13. What is the difference between Error and Exception in C#?

An exception is an Object of a type deriving from the System.Exception class. SystemException is thrown by the CLR (Common Language Runtime) when errors occur that are nonfatal and recoverable by user programs.

Exception syntax:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try    {    //write your code here    }    Catch (exception type)    {    //writ your code here    } |

Errors are unchecked exception and is difficult to handle in code.

### 14. Why to use Lock statement in C#?

Using Lock statement you can ensure only one thread can be executed at any point of time.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | using System;  using System.Threading;  using System.Threading.Tasks;  namespace RaceCondition  {      class Program      {          static object locker = new object();          private static int counter;          static void Main(string[] args)          {              new Thread(PrintStar).Start();              new Thread(PrintPlus).Start();          }            static void PrintStar()          {              lock (locker) // Thread safe code              {                  for (counter = 0; counter < 5; counter++)                  {                      Console.Write(" \* " + "\t");                  }              }          }            static void PrintPlus()          {              lock (locker) // Thread safe code              {                  for (counter = 0; counter < 5; counter++)                  {                      Console.Write(" + " + "\t");                  }              }          }      }  } |

### 15. What is the difference between Yield and Return in C#?

#### **return statement in C#:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static int SimpleReturn()  {      return 1;      return 2;      return 3;  }  static void Main(string[] args)  {      Console.WriteLine(SimpleReturn());      Console.WriteLine(SimpleReturn());      Console.WriteLine(SimpleReturn());      Console.WriteLine(SimpleReturn());  } |

Output:  
1  
1  
1  
1  
In above example int SimpleReturn function three returns are mentioned but no matter how many times this function is called from anywhere, here from main the function will always return first value i.e 1.

#### **Yield statement in C#:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static IEnumerable<int> YieldReturn()  {      yield return 1;      yield return 2;      yield return 3;  }  static void Main(string[] args)  {      foreach (int i in YieldReturn())      {          Console.WriteLine(i);      }  } |

Output:  
1  
2  
3  
The only difference between yield and return is whenever yield statement is encountered in a function, the execution of function is suspended and a value is send back to the caller but because of yield whenever the function is called again, the execution of function begin where it left off previously. When resumed, the function continues execution immediately after the last yield run. Thus yield allows a function to produce a series of values over time. The only requirement for yield return statement is that the function containing yield should return an IEnumerable and no matter from where that function is called it should be called from an iteration block i.e foreach statement.

### ¿Que es inyección de dependencias?

Basicamente es quitarle la responsabilidad a nuestra clase principal de crear instancias a sus dependencias. En pocas palabras, en vez de que nuestra clase haga una instancia internamente a otra clase que depende, mejor le pasamos la clase que va a implementar como un parámetro adicional, de esta manera evitamos la dependencia directa. ¿Se entendió?, bueno ... espero que con el ejemplo se entienda mejor.

### Nuestro ejemplo

En nuestro ejemplo vamos a suministrar a un soldado que arma debería usar en el combate. Para ello, hemos creado las siguientes clases:

#### Las armas convertidas a Clase

public class Revolver

{

public string Disparar()

{

return "Pum Pum ..";

}

}

public class Rifle

{

public string Disparar()

{

return "Pum pum pum pum pum ..";

}

}

public class Escopeta

{

public string Disparar()

{

return "pum PUMMM !! ..";

}

}

Como se darán cuenta, las armas implementan el método**Disparar()**, la cual el disparo varía dependiendo del tipo de arma que se elija.

#### Nuestra clase Soldado

Esta clase implementa los métodos necesarios para realizar los disparos para cada arma.

public class Soldado

{

public string DispararRevolver()

{

return new Revolver().Disparar();

}

public string DispararRifle()

{

return new Rifle().Disparar();

}

public string DispararEscopeta()

{

return new Escopeta().Disparar();

}

}

* **¿Cual es el problema?,**que la dependencia la tiene que implementar el mismo Soldado.
* **¿Y eso en que nos afecta?**
  + Vamos a tener que modificar nuestra clase Soldado en un futuro si queremos agregar más armas, ahora esta simple porque solo hay 3 armas y cada una tiene un método. ¿Pero si fuerán 200 armas y cada una tiene 100 métodos?.
  + Realizar una prueba unitaria sería bastante tedioso, hay que probar método por método.
  + La clase a futuro va a ser un dolor de cabeza, teniendo un código que no permita su mantenimiento/mejora a futuro.
  + Al momento de instanciar la clase, vamos a tener que **llamar a c/u de los métodos**para manipular el arma seleccionada. Eso nos daría un serio problema, porque tendríamos que **modificar el código constantemente** si es que quisieramos cambiar de arma o hacer un uso **excesivo y vulgar de los IF o SWITCH para saber con que arma se debe trabajar**.

### ¿Cual es la solución?

Implementar una **interface** yque el **constructor de la clase Soldado la reciba** como parámetro.

#### Paso #1

Crear la interface y hacer que las armas implementen dicha interface.

public interface IArma {

string Disparar();

}

public class Revolver : IArma

{

public string Disparar()

{

return "Pum Pum ..";

}

}

public class Rifle : IArma

{

public string Disparar()

{

return "Pum pum pum pum pum ..";

}

}

public class Escopeta : IArma

{

public string Disparar()

{

return "pum PUMMM !! ..";

}

}

#### Paso #2

Modificar nuestra clase Soldado para que reciba dicha interface.

public class Soldado

{

protected IArma arma;

public Soldado(IArma \_arma)

{

this.arma = \_arma;

}

public string Disparar()

{

return this.arma.Disparar();

}

}

Si se dan cuenta ahora, solo tenemos un **método "Disparar"**. Nuestra clase Soldado sabe que debe disparar un Arma, es nuestro constructor el encargado de setear que arma debe usar.

Veamos como se instancia nuestra clase Soldado usando como arma principal el Revolver:

var soldado = new Soldado(new Revolver());

soldado.Disparar();

Si queremos usar la escopeta:

var soldado = new Soldado(new Escopeta());

soldado.Disparar();

¿Y el rifle?

var soldado = new Soldado(new Rifle());

soldado.Disparar();

¿Se dieron cuenta lo lindo que ha quedado nuestro código?. Al final lo que hemos hecho es eliminar la dependenia de una clase dentro de otra clase.

**NOTA**: Me imagino que si has comprendido el ejemplo, tu mente ahora procesa varios casos para la cual usaría la **inyección de dependencia.**Sino fuera así, deja un comentario y te puedo orientar.

**Dependency property**: A property that is backed by a [**DependencyProperty**](http://msdn.microsoft.com/en-us/library/system.windows.dependencyproperty.aspx).

The purpose of dependency properties is to provide a way to compute the value of a property **based** on the **value of other inputs**.

These other inputs might include system properties such as themes and user preference, just-in-time property determination mechanisms such as data binding and animations/storyboards, multiple-use templates such as resources and styles, or values known through parent-child relationships with other elements in the element tree.

KeyWords

* C#
* C++
* Java
* SQL

Interviews Questions:

1. Experience and Importance of the support, testing and develop procedures.
2. How to solve the problems of replication between two Databases.
   * Pooling or Observer
3. What is a BTree.

. Arbol balanceado:  a **B-tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The **B-tree** is a generalization of a binary search tree in that a node can have more than two children.

Uses: Index in data bases.

1. Stateless and statefull protocol, what type is HTTP?
   1. Stateless: Sin estado. Asincrono. HTTP.
   2. Statefull: Con estado. Syncrono. FTP.
2. What is AJAX and uses?
   1. **JavaScript Asíncrono y XML (AJAX)** no es una tecnología por sí misma, es un término que describe un nuevo modo de utilizar conjuntamente varias tecnologías existentes. Esto incluye: [HTML](https://developer.mozilla.org/es/docs/Web/Guide/es/HTML) o [XHTML](https://developer.mozilla.org/es/docs/Web/Guide/es/XHTML), [CSS](https://developer.mozilla.org/es/docs/Web/Guide/es/CSS), [JavaScript](https://developer.mozilla.org/es/docs/Web/Guide/es/JavaScript), [DOM](https://developer.mozilla.org/es/docs/Web/Guide/es/DOM), [XML](https://developer.mozilla.org/es/docs/Web/Guide/es/XML), [XSLT](https://developer.mozilla.org/es/docs/Web/Guide/es/XSLT), y el objeto [XMLHttpRequest](https://developer.mozilla.org/es/docs/Web/Guide/es/XMLHttpRequest). Cuando estas tecnologías se combinan en un modelo AJAX, es posible lograr aplicaciones web capaces de actualizarse continuamente sin tener que volver a cargar la página completa. Esto crea aplicaciones más rápidas y con mejor respuesta a las acciones del usuario.
3. Desgin Pattern: Singleton, Observe.
   1. Singleton:

A class of which only a single instance can exist and it’s globaly accessible. Methods: static Singleton getInstance(), the constructor is private.

* 1. Observe:

A way of notifying change to a number of classes. Communicate the changes in one class to the reset connected. 2 Interfaces: ISubject and IObserver. Methods: ISubject.AttachObserber(), ISubject.DettachObserber() and ISubject.SendNotification().

* 1. Factory- Abstract Factory:

The abstract class or interface constructor must be protected. Used in the List of the User Interface to group the same family. Abstract-Factory = Factory of factories (grouping different families). getType().

1. Create a small numerical function.

Desgin Pattern:

# Design Patterns

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

### Uses of Design Patterns

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

### [Creational design patterns](https://sourcemaking.com/design_patterns/creational_patterns)

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

[](https://sourcemaking.com/design_patterns/abstract_factory)

* [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
  Creates an instance of several families of classes
* [**Builder**](https://sourcemaking.com/design_patterns/builder)  
  Separates object construction from its representation
* [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)  
  Creates an instance of several derived classes
* [**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
* [**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
  A fully initialized instance to be copied or cloned
* [**Singleton**](https://sourcemaking.com/design_patterns/singleton)  
  A class of which only a single instance can exist

### [Structural design patterns](https://sourcemaking.com/design_patterns/structural_patterns)

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

[](https://sourcemaking.com/design_patterns/decorator)

* [**Adapter**](https://sourcemaking.com/design_patterns/adapter)  
  Match interfaces of different classes
* [**Bridge**](https://sourcemaking.com/design_patterns/bridge)  
  Separates an object’s interface from its implementation
* [**Composite**](https://sourcemaking.com/design_patterns/composite)  
  A tree structure of simple and composite objects
* [**Decorator**](https://sourcemaking.com/design_patterns/decorator)  
  Add responsibilities to objects dynamically
* [**Facade**](https://sourcemaking.com/design_patterns/facade)  
  A single class that represents an entire subsystem
* [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight)  
  A fine-grained instance used for efficient sharing
* **[](https://sourcemaking.com/design_patterns/proxy)**

[**Private Class Data**](https://sourcemaking.com/design_patterns/private_class_data)  
Restricts accessor/mutator access

* [**Proxy**](https://sourcemaking.com/design_patterns/proxy)  
  An object representing another object

### [Behavioral design patterns](https://sourcemaking.com/design_patterns/behavioral_patterns)

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

[](https://sourcemaking.com/design_patterns/interpreter)

* [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility)  
  A way of passing a request between a chain of objects
* [**Command**](https://sourcemaking.com/design_patterns/command)  
  Encapsulate a command request as an object
* [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter)  
  A way to include language elements in a program
* [**Iterator**](https://sourcemaking.com/design_patterns/iterator)  
  Sequentially access the elements of a collection
* [**Mediator**](https://sourcemaking.com/design_patterns/mediator)  
  Defines simplified communication between classes
* [**Memento**](https://sourcemaking.com/design_patterns/memento)  
  Capture and restore an object's internal state
* [**Null Object**](https://sourcemaking.com/design_patterns/null_object)  
  Designed to act as a default value of an object
* [**Observer**](https://sourcemaking.com/design_patterns/observer)  
  A way of notifying change to a number of classes
* **[](https://sourcemaking.com/design_patterns/state)**

[**State**](https://sourcemaking.com/design_patterns/state)  
Alter an object's behavior when its state changes

* [**Strategy**](https://sourcemaking.com/design_patterns/strategy)  
  Encapsulates an algorithm inside a class
* [**Template method**](https://sourcemaking.com/design_patterns/template_method)  
  Defer the exact steps of an algorithm to a subclass
* [**Visitor**](https://sourcemaking.com/design_patterns/visitor)  
  Defines a new operation to a class without change

### Criticism

The concept of design patterns has been criticized by some in the field of computer science.

#### Targets the wrong problem

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay [**Revenge of the Nerds**](http://www.paulgraham.com/icad.html).

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

#### Lacks formal foundations

The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. AtOOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.

#### Leads to inefficient solutions

The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

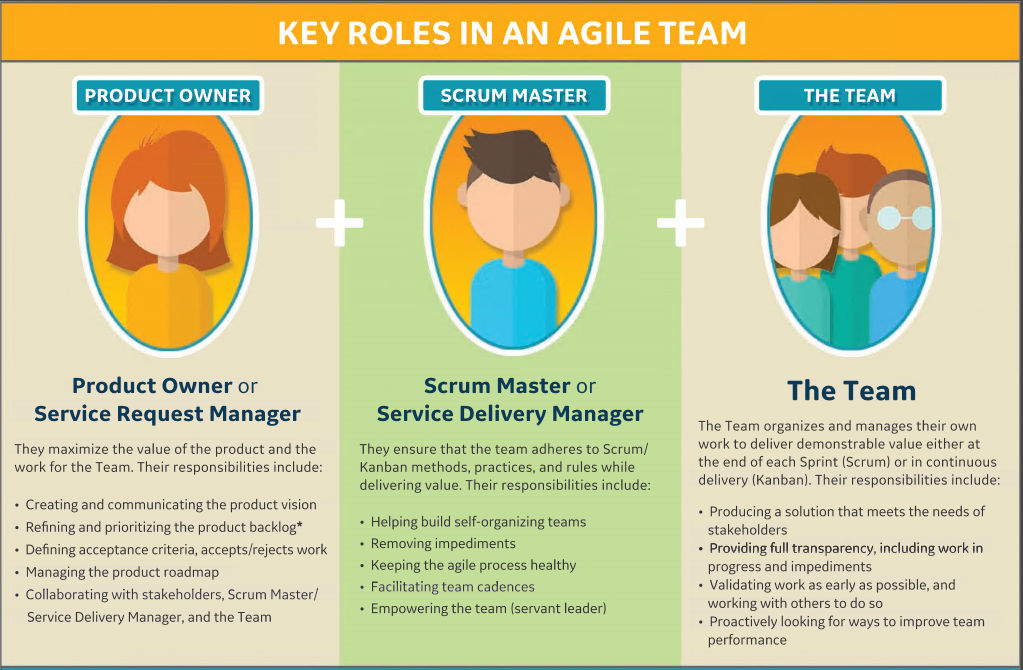
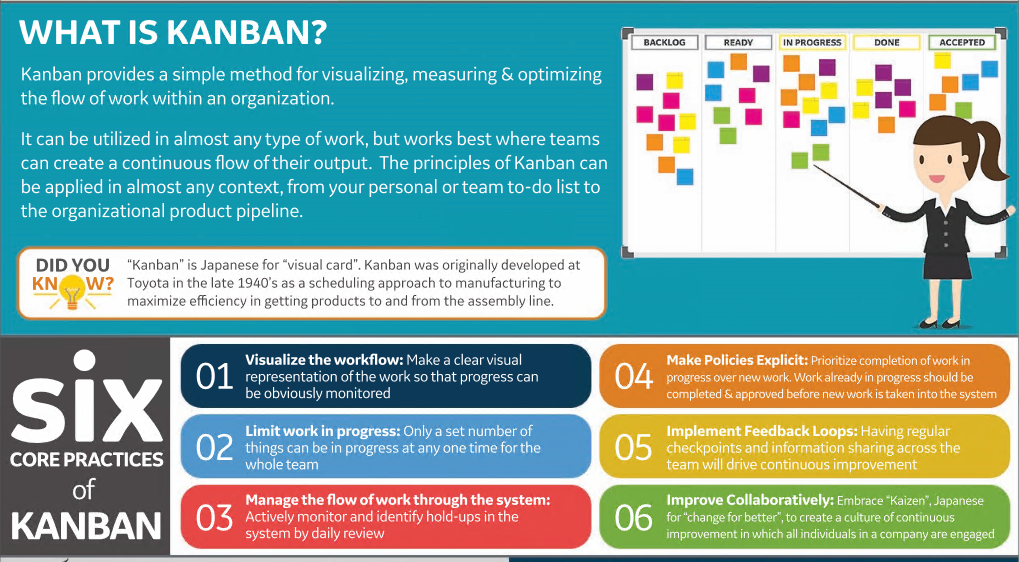
#### Does not differ significantly from other abstractions

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

<https://sourcemaking.com/design_patterns>

# Agile

# 

# SOLID

https://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp