LUCKIA

Capacitación .NET

Propuesta de contenido de curso formativo

Este documento contiene información y material confidencial propiedad de Tecnologías Plexus, S.L. Los materiales, ideas y conceptos contenidos en esta propuesta serán utilizados exclusivamente para evaluar las capacidades de la empresa y no deberán ser divulgados fuera de su organización o utilizados con propósitos distintos a los mencionados. No está permitida su reproducción total o parcial, ni su uso con otras organizaciones para ningún otro propósito, excepto autorización previa por escrito.

luckia PLEXUS

Índice

1. Introducción a la programación 5
1.1. Diseño de Algoritmos 5
1.1.1. Definición del algoritmo 5
1.1.2. Etapa del diseño 5
1.1.3. Herramientas para representar algoritmos 6
1.2. Introducción al Pseudocódigo 6
1.2.1. Estructura del Algoritmo 6
1.3. Pseudocódigo: Estructuras alternativas 8
1.3.1. Estructura Si 8
1.3.2. Estructura Segun 8
1.4. Pseudocódigo: Estructuras Repetitivas 9
1.4.1. Estructura Mientras 9
1.4.2. Estructura Repetir-Hasta-Que 9
1.4.3. Estructura Para 10
1.5. Pseudocódigo: Arreglos 10
1.6. Pseudocódigo: Programación estructurada 11
1.6.1. Funciones 11
1.6.2. Procedimientos 12
2. Visual Studio 2017: IDE Desarrollo 13
2.1. Ventana de Inicio 13
2.2. Creación de proyecto: Arquetipos y proc. batchs. 14
2.2.1. Batchs 14
2.3. Administración de paquetes NuGet 15
2.4. Explorador de soluciones 16
2.5. Debugs y Breakpoints 16
2.6. Team Explorer 17
2.7. Lista de Errores 17
2.8. Ventana de Resultados 18
3. POO 19
3.1. Introducción a la POO 19
3.1.1. Objetos y Clases 19
3.1.2. Requerimientos funcionales y no funcionales 20
3.2. Fundamentos de C# y POO 20
3.2.1. Campos y Propiedades 20
3.2.2. Constructores 21
3.2.3. Signaturas y Sobrecargas 21
3.2.4. Interfaces 22
3.2.5. Clases Estáticas 23
3.3. Herencia 24
3.3.1. Introducción a Herencia 24
3.4. Modificadores de Acceso 25
3.5. Los 4 pilares de la Programación Orientada a Objetos 26
3.6. Complejidad de clases 27
3.7. Principios SOLID 28
3.7.1. Principio de responsabilidad única 28
3.7.2. Principio de abierto/cerrado 29
3.7.3. Principio de sustitución de Liskov 29
3.7.4. Principio de segregación de interfaces 31
3.7.5. Principio de inversión de dependencias 32

4. Fundamentos de UML 34
4.1. Introducción a UML 34
4.2. Casos de uso 34
4.3. Diagramas 36
4.3.1. Diagrama de secuencia 36
4.3.2. Diagrama de comunicación/colaboración 36
4.3.3. Diagrama de clases 37
4.3.4. Diagrama de estado 39
5. Arquitectura de aplicaciones 40
5.1. Arquitectura de N-Capas 40
5.2. Patrones de diseño 42
5.2.1. ¿Qué son los Patrones de Diseño? 42
5.2.2. Patrones de diseño creacionales 42
5.2.3. Patrones de diseño estructurales 44
5.2.4. Patrones de comportamiento 47
5.3. Inyección de dependencias 50
5.3.1. Introducción 50
5.3.2. DI con Ioc Autofac 52
6. Introducción al Desarrollo WEB 54
6.1. Principales características de un desarrollo WEB 54
6.2. Lado Cliente/Lado Servidor 54
7. SQL 55
7.1. Introducción 55
7.1.1. Comandos 55
7.1.2. Claves 56
7.1.3. Clausulas 57
7.1.4. Operadores 57
7.1.5. Funciones de Agregado 57
7.2. Obtención de datos 58
7.2.1. Ordenación de registros: 58
7.2.2. Consulta con predicado: 58
7.2.3. Subconsultas 58
7.2.4. Consultas de Unión Internas: 59
7.3. Diagrama ER 60
7.3.1. Introducción 60
7.3.2. Componentes del diagrama ER 61
8. Repositorio ..... ¡Error! Marcador no definido.
8.1. Conceptos Básicos ..... **¡Error! Marcador no definido.**
8.2. GitLab ..... **¡Error! Marcador no definido.**
8.3. IC ,EC y DC: Teamcity & Octopus ..... **¡Error! Marcador no definido.**
8.3.1. Integración Continua y Despliegue Continuo ..... **¡Error! Marcador no definido.**
8.3.2. TeamCity ..... **¡Error! Marcador no definido.**
8.3.3. Octopus ..... **¡Error! Marcador no definido.**
9. C# ..... ¡Error! Marcador no definido.
9.1. Descubriendo .NET ..... **¡Error! Marcador no definido.**
9.2. Fundamentos C# ..... **¡Error! Marcador no definido.**
9.2.1. Introducción e historia C# ..... **¡Error! Marcador no definido.**
9.2.2. NET Framework 4.6.1, CLR y FCL ..... **¡Error! Marcador no definido.**
9.2.3. Variables, constantes ..... **¡Error! Marcador no definido.**
9.2.4. Operadores y expresiones ..... **¡Error! Marcador no definido.**
9.2.5. Estructuras de control ..... **¡Error! Marcador no definido.**
9.2.6. Enumeraciones ..... **¡Error! Marcador no definido.**

9.2.7. Estructuras ..... ¡Error! Marcador no definido.
9.2.8. Métodos en C# ..... ¡Error! Marcador no definido.
9.2.9. Manejo de Excepciones ..... ¡Error! Marcador no definido.
9.2.10. Clases y Objetos ..... ¡Error! Marcador no definido.
9.3. Threads ..... ¡Error! Marcador no definido.
9.3.1. Introducción. ¿Qué son los hilos? ..... ¡Error! Marcador no definido.
9.3.2. Creación y administración de hilos ..... ¡Error! Marcador no definido.
9.4. WPF ..... ¡Error! Marcador no definido.
9.4.1. Introducción a WPF ..... ¡Error! Marcador no definido.
9.4.2. Controles ..... ¡Error! Marcador no definido.
9.4.3. Layouts ..... ¡Error! Marcador no definido.
9.4.4. Propiedades y eventos ..... ¡Error! Marcador no definido.
9.4.5. Databinding ..... ¡Error! Marcador no definido.
9.4.6. Resources ..... ¡Error! Marcador no definido.
9.4.7. Triggers,Templates,Dialogs ..... ¡Error! Marcador no definido.
9.5. MVC ..... ¡Error! Marcador no definido.
9.5.1. Introducción al MVC ..... ¡Error! Marcador no definido.
9.5.2. Estructura MVC ..... ¡Error! Marcador no definido.
9.5.3. Razor Views ..... ¡Error! Marcador no definido.
9.6. ORMs ..... ¡Error! Marcador no definido.
9.6.1. Introducción ..... ¡Error! Marcador no definido.
9.6.2. NHiberbate & ActiveRecord ..... ¡Error! Marcador no definido.
9.6.3. EntityFramework ..... ¡Error! Marcador no definido.
9.7. API ..... ¡Error! Marcador no definido.
9.7.1. Concepto de API ..... ¡Error! Marcador no definido.
9.7.2. API REST con C# ..... ¡Error! Marcador no definido.
9.7.3. SOAP con C# ..... ¡Error! Marcador no definido.
9.7.4. OAuth 2.0 ..... ¡Error! Marcador no definido.
9.8. SignalR ..... ¡Error! Marcador no definido.
9.8.1. Introducción a SignalR ..... ¡Error! Marcador no definido.

1. Introducción a la programación

Cuando se plantea una petición/tarea que solicita un desarrollo informático, se requiere cumplir ciertas fases necesarias para el correcto planteamiento de la solución al problema.

Dichas fases son: el análisis del problema, el diseño del algoritmo y finalmente la codificación.

1.1. Diseño de Algoritmos

1.1.1. Definición del algoritmo

Un algoritmo es un conjunto de acciones que especifican la secuencia de operaciones realizar, en orden, para resolver un problema.

Los algoritmos son independientes tanto del lenguaje de programación como del ordenador que los ejecuta.

Las características de los algoritmos son:

- Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
- Un algoritmo debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

1.1.2. Etapa del diseño

Aunque la soluciones a problemas más complejos pueden requerir muchos más pasos, las estrategias seguidas usualmente a la hora de encontrar algoritmos para problemas complejos son:

- Partición o divide y vencerás: consiste en dividir un problema grande en unidades más pequeñas que puedan ser resueltas individualmente.

Ejemplo: Podemos dividir el problema de limpiar una casa en labores más simple correspondientes a limpiar cada habitación.

La descomposición del problema original en subproblemas más simples y a continuación dividir estos subproblemas en otros más simples se denomina diseño descendente (top-down design)

- Resolución por analogía/reutilización: Dado un problema, se trata de recordar algún problema similar que ya esté resuelto. Los dos problemas análogos pueden incluso pertenecer áreas de conocimiento totalmente distintas.

Ejemplo: El cálculo de la media de las temperaturas de las provincias andaluzas y la media de las notas de los alumnos de una clase se realiza del mismo modo.

Normalmente, tras la primera descripción del problema (poco específica), se realiza una siguiente descripción más detallada con más pasos concretos. Este proceso se denomina refinamiento del algoritmo.

Ejemplo de diseño de algoritmo:

Leer el radio de un circunferencia y calcular e imprimir su superficie y su circunferencia.

- Se puede dividir en tres subproblemas más sencillos:
  - Leer Radio
  - Calcular Superficie
  - Calcular Longitud
  - Escribir resultados
- Refinamiento del algoritmo:
  - Leer Radio
  - Superficie <- PI * Radio ^ 2
  - Longitud <- 2 * PI * Radio
  - Escribir Radio, Longitud, Superficie

1.1.3. Herramientas para representar algoritmos

- Un diagrama de flujo es una de las técnicas de representación gráfica de algoritmos más antiguas. Ventajas: permite altos niveles de estructuración y modularización y es fácil de usar. Desventajas: son difíciles de actualizar y se complican cuando el algoritmo es grande.
- El pseudocódigo, nos permite una aproximación del algoritmo al lenguaje natural y por tanto un a redacción rápida del mismo.

1.2. Introducción al Pseudocódigo

1.2.1. Estructura del Algoritmo

- Comienza con la palabra clave Proceso (o alternativamente Algoritmo, son sinónimos) seguida del nombre del programa.
- Le sigue una secuencia (Estructura de control secuencial) de instrucciones. Una secuencia de instrucciones es una lista de una o más instrucciones y/o estructuras de control.
- Finaliza con la palabra FinProceso (o FinAlgoritmo).
- La identación no es significativa, pero se recomienda para que el código sea más legible.
- No se diferencia entre mayúsculas y minúsculas. Preferible las minúsculas, aunque a veces se añaden automáticamente los nombres con la primera letra en mayúsculas.
- Se pueden introducir comentarios después de una instrucción, o en líneas separadas, mediante el uso de la doble barra (//)

```
//Leer el radio de un círculo y calcular e imprimir su superficie y su
circunferencia.
//Análisis
//Entradas: Radio del circulo (Variable RADIO).
//Salidas: Superficie del circulo (Variable SUPERFICIE) y Circunferencia del
circulo (Variable PERIMETRO)
//Variables: RADIO, SUPERFICIE, PERIMETRO de tipo REAL

Proceso Circulo
    Definir radio,superficie,perimetro como Real;
    Escribir "Introduce el radio de la circunferencia:";
    Leer radio;
    superficie <- PI * radio ^ 2;
    perimetro <- 2 * PI * radio;
    Escribir "La superficie es ",superficie;
    Escribir "El perímetro es ",perimetro;
FinProceso
```

Tipo de datos simples usados:

- Números enteros: Representan números enteros.
- Números reales: Representan números reales.
- Cadenas de caracteres: Representan cadenas de caracteres.
- Valores lógicos: Representan valores lógicos.

Variables:

Tal y como se aprecia en el código de ejemplo, se crean con la palabra “Definir” y pueden crearse varias a la vez separándolas por comas. Con la palabra “Como” podemos indicar el tipo de dato de la variable.

```
Definir numero1, numero2 como Entero;
Definir superficie, perimetro como Real;
Definir nombre como Caracter;
Definir mayor_edad como Logico;
```

Usaremos en nuestros casos el operador ‘<-‘ como asignación de valor a variables para evitar la confusión con el comparador de igualdad ‘=’

Operadores:

- Operadores aritméticos: +, -, *, /, (% ó mod), ^
- Operadores de comparación: >, <, >=, <=, =, ¡=
- Operadores lógicos: (& ó Y), (| ó O), (~ ó NO)

Entrada y salida de información:

Usaremos las palabras Leer y Escribir para obtener información y mostrar información

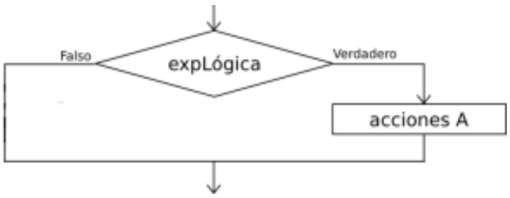
1.3. Pseudocódigo: Estructuras alternativas

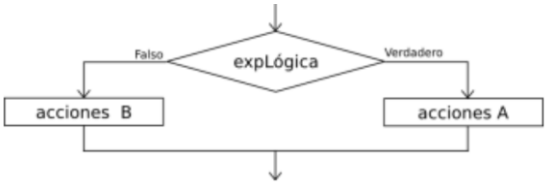
1.3.1. Estructura Si

Al ejecutarse la instrucción y evaluarse la condición lógica, se ejecuta un bloque de sentencias u otro en función de su valor verdadero o falso.

```
Proceso mayor_edad
    Definir edad como entero;
    Escribir "Dime tu edad:";
    Leer edad;
    Si edad>=18 Entonces
        Escribir "Eres mayor de edad";
    FinSi
    Escribir "Programa terminado";
FinProceso
```

```
Proceso mayor_edad
    Definir edad como entero;
    Escribir "Dime tu edad:";
    Leer edad;
    Si edad>=18 Entonces
        Escribir "Eres mayor de edad";
    SiNo
        Escribir "Eres menor de edad";
    FinSi
    Escribir "Programa terminado";
FinProceso
```

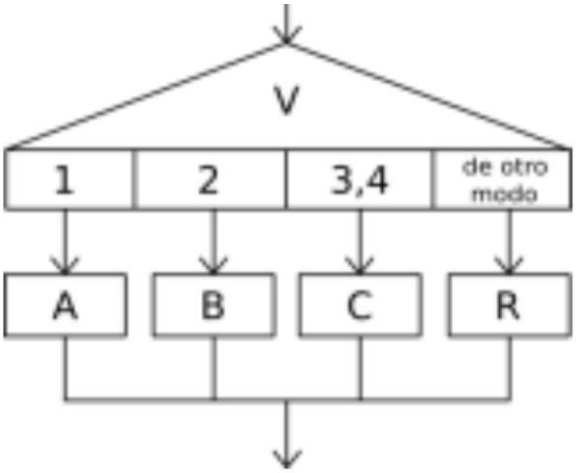




1.3.2. Estructura Segun

La secuencia de instrucciones ejecutada por una instrucción Segun depende del valor de una variable numérica.

```
Proceso notas
    Definir nota como entero;
    Escribir "Dime tu nota:";
    Leer nota;
    Segun nota Hacer
        1,2,3,4: Escribir "Suspenso";
        5: Escribir "Suficiente";
        6,7: Escribir "Bien";
        8: Escribir "Notable";
        9,10: Escribir "Sobresaliente";
        De Otro Modo:
            Escribir "Nota incorrecta";
    FinSegun
    Escribir "Programa terminado";
FinProceso
```



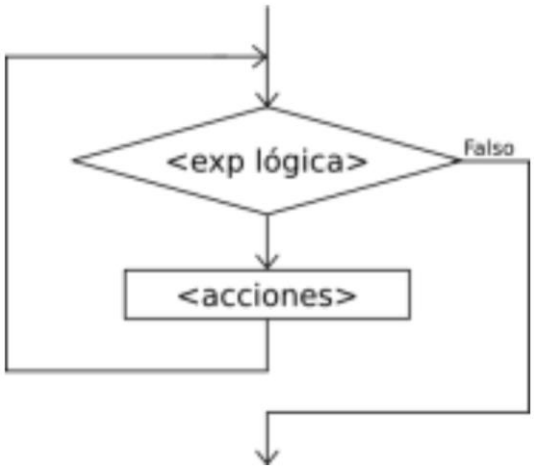
1.4. Pseudocódigo: Estructuras Repetitivas

1.4.1. Estructura Mientras

La instrucción Mientras ejecuta una secuencia de instrucciones mientras una condición sea verdadera.

A la repetición de instrucciones se denomina bucle y a cada una de las ejecuciones del bloque de instrucciones se denomina iteración.

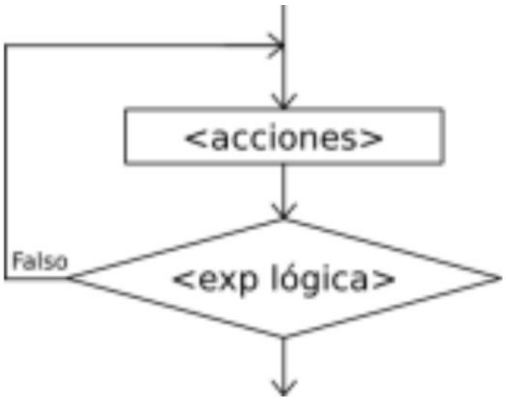
```
Proceso login
    Definir secreto, clave como cadena;
    secreto <- "asdasd";
    Escribir "Dime la clave:";
    Leer clave;
    Mientras clave<>secreto Hacer
        Escribir "Clave incorrecta!!!";
        Escribir "Dime la clave:";
        Leer clave;
    FinMientras
    Escribir "Bienvenido!!!";
    Escribir "Programa terminado";
FinProceso
```



1.4.2. Estructura Repetir-Hasta-Que

La instrucción Repetir-Hasta Que ejecuta una secuencia de instrucciones hasta que la condición sea verdadera.

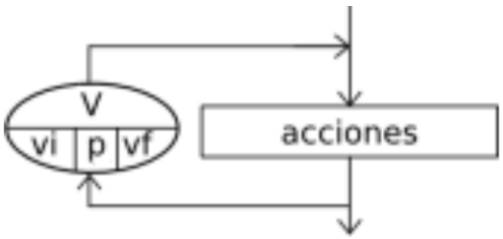
```
Proceso login
    Definir secreto, clave como cadena;
    secreto <- "asdasd";
    Repetir
        Escribir "Dime la clave:";
        Leer clave;
        Si clave<>secreto Entonces
            Escribir "Clave incorrecta!!!";
        FinSi
    Hasta Que clave=secreto
    Escribir "Bienvenido!!!";
    Escribir "Programa terminado";
FinProceso
```



1.4.3. Estructura Para

La instrucción Para ejecuta una secuencia de instrucciones un número determinado de veces.

```
Proceso Contar
    Definir var como Entero;
    Para var<-1 Hasta 10 Hacer
        Escribir Sin Saltar var," ";
    FinPara
FinProceso
```



1.5. Pseudocódigo: Arreglos

Un array (o arreglo) es una estructura de datos con elementos homogéneos, del mismo tipo, numérico o alfanumérico, reconocidos por un nombre en común. Para referirnos a cada elemento del array usaremos un índice (empezamos a contar por 0).

Existen arrays de una dimensión, denominados vector y arrays de dos dimensiones denominados tablas.

Ejemplo: Definir un array de una dimensión (también llamado vector) de 10 elementos enteros. Para acceder a algún elemento del mismo usaremos vector[indice]

```
Definir vector como Entero;
Dimension vector[10];
```

Ejemplo: Definir una array de dos dimensiones (también llamado matriz o tabla) de 3 filas y cuatro columnas de cadenas. Para acceder a algún elemento del mismo usaremos tabla[indice1,indice2]

```
Definir tabla como Cadenas;
Dimension tabla[3,4];
```

1.6. Pseudocódigo: Programación estructurada

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa o algoritmo, utilizando únicamente subrutinas (funciones o procedimientos) y tres estructuras: secuencia, alternativas y repetitivas.

Aplicando esta programación modular conseguimos, además de hacer más legible y mantenible un programa, la reutilización, ordenación y eficiencia del código.

Cuando usamos subrutinas (funciones o procedimientos), debemos tener claro el ámbito de las variables: Una variable creada en el cuerpo de nuestro programa NO será visible para una subrutina, al igual que una variable creada en una subrutina sólo será visible dentro de dicha subrutina. Para comunicarnos con el exterior, haremos uso de los parámetros, que son valores que se pasan a las subrutinas en forma de variables.

1.6.1. Funciones

Las funciones son unas subrutinas que devuelven un valor. Se declara con la palabra clave Function, seguida de la variable de retorno, el signo de asignación, el nombre del subproceso, y finalmente, la lista de argumentos entre paréntesis.

```
Funcion max <- CalcularMaximo(num1,num2)
    Definir max Como Entero;
    Si num1>num2 Entonces
        max <- num1;
    SiNo
        max <- num2;
    FinSi
FinFuncion
```

Para usar una función, debemos llamarla en el cuerpo de nuestro programa, de la siguiente manera:

```
num1 <- CalcularMaximo(5,6)
```

Funciones recursivas:

Una función recursiva es aquella que al ejecutarse hace llamadas a ella misma. Por lo tanto tenemos que tener “un caso base” que hace terminar el bucle de llamadas.

```
Funcion fact <- CalcularFactorial(num)
    Definir fact Como Entero;
    Si num=0 O num=1 Entonces
        fact <- 1;
    SiNo
        fact <- num * CalcularFactorial(num-1);
    FinSi
FinFuncion

Proceso ProgramaPrincipal
    Escribir "El factorial de 6 es ",CalcularFactorial(6);
FinProceso
```

1.6.2. Procedimientos

Los procedimientos son unas subrutinas que ejecutan alguna tarea concreta pero que a diferencia de las funciones, no devuelven ningún dato.

Comienza con la palabra clave SubProceso , seguida de la lista de argumentos si los tuviese entre paréntesis.

```
Proceso Titulos
    Definir titulo como cadena;
    titulo <- "Ejercicio 1";
    Subrayar(titulo);
    Escribir "";
FinProceso
```

Para usar un procedimiento, debemos llamarlo en el cuerpo de nuestro programa, de la siguiente manera:

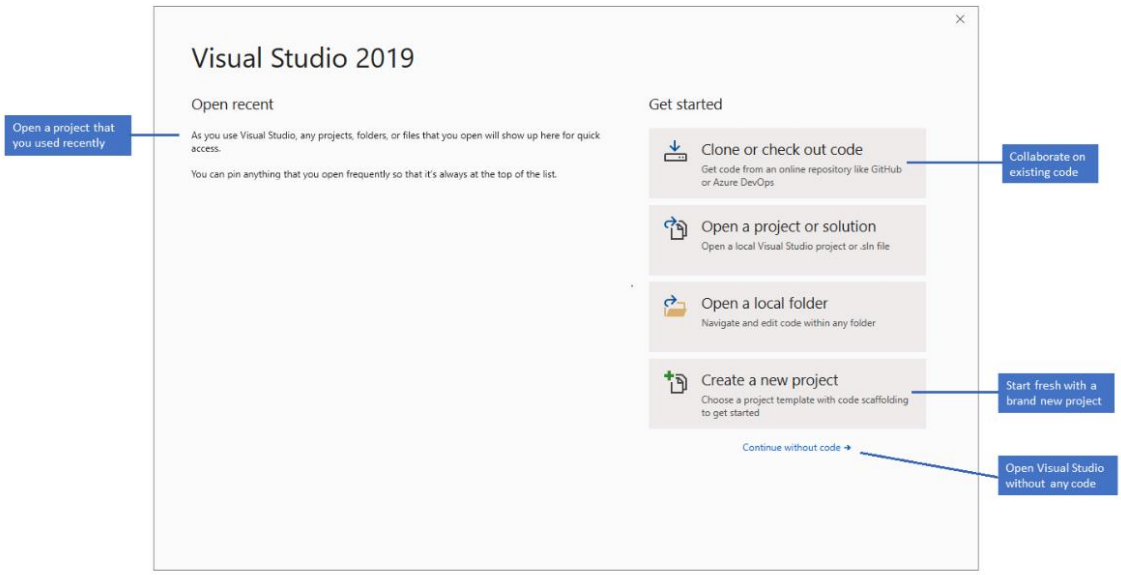
```
Escribir CalcularMaximo(1,2)
```

2. Visual Studio 2017: IDE Desarrollo

Un entorno de desarrollo integrado o entorno de desarrollo interactivo, (Integrated Development Environment (IDE)), es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software. Normalmente, un IDE consiste de un editor de código fuente, herramientas de construcción automáticas y un depurador.
La mayoría de los IDE tienen auto-completado inteligente de código (IntelliSense). Además de un compilador, un intérprete, o ambos.
El ID de desarrollo que usaremos en nuestro caso será Visual Studio 2017. A continuación, realizaremos un breve repaso por las funcionalidades más básicas del mismo:

2.1. Ventana de Inicio.

Lo primero que se ve al abrir Visual Studio es la ventana de inicio. La ventana de inicio está concebida para ayudar a "obtener código" más rápido. Tiene opciones para clonar o desproteger código, abrir una solución o un proyecto existente, crear un nuevo proyecto o simplemente abrir una carpeta que contiene algunos archivos de código.

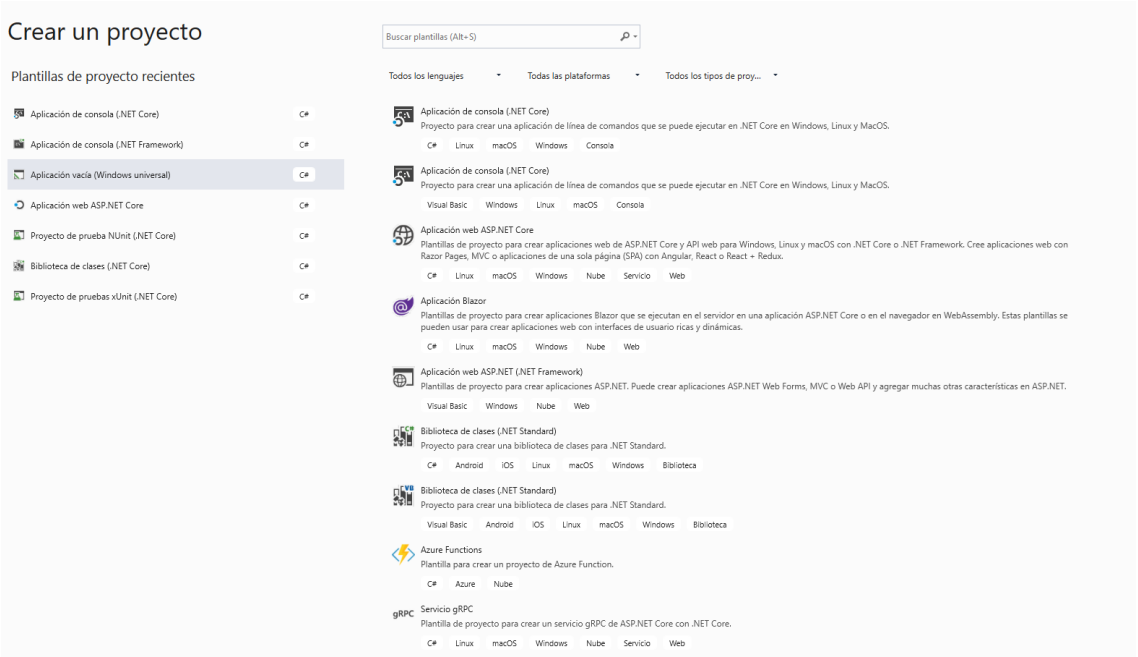


Si es la primera vez que se usa Visual Studio, la lista de proyectos recientes está vacía. Podría trabajar en local, con código base no compartido o bien puede conectarse a un repositorio para, o bien crear un nuevo proyecto o clonar un proyecto de un proveedor de origen como GitHub o Azure DevOps. Veremos más info de esto en el capítulo “Repositorio: GIT”.
La opción Continuar sin código simplemente abre el entorno de desarrollo de Visual Studio sin ningún proyecto o código concreto cargado.

2.2. Creación de proyecto: Arquetipos y proc. batchs.

En la ventana de inicio, al elegir Crear un proyecto nuevo, Se abre un cuadro de diálogo de nombre Crear un proyecto. Aquí, puede buscar, filtrar y seleccionar una plantilla de proyecto. También muestra una lista de las plantillas de proyecto recientemente usadas.

En el cuadro de búsqueda superior, se pueden filtrar los tipos de proyecto.



Seleccionaremos una de las plantillas listadas y, luego, elija Siguiente. En la página Configure su nuevo proyecto, acepte el nombre y la ubicación predeterminados del proyecto y haga clic en Crear. En este momento se crea el proyecto y se abren los ficheros autogenerados en el explorador de soluiones de Visual Studio.

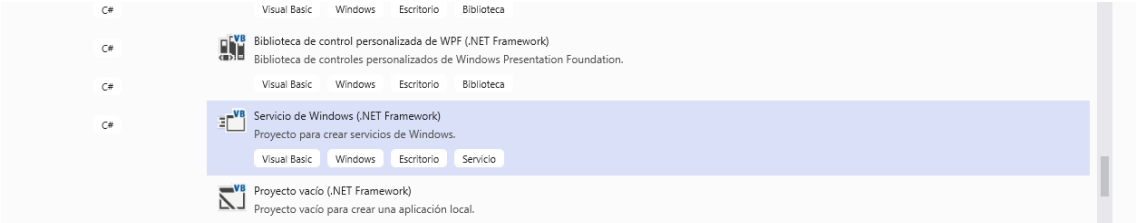
2.2.1. Batchs

Un proceso batch es un elemento que llevar a cabo una operación particular de forma automática..

Por ejemplo, un programa que convierta un grupo de todas las imágenes a la vez de un formato a otro.

Un proceso batch puede tener una planificación periódico y puede ser gestionado/manipulado a travñes del visor de servicios.

Para crear un proyecto batch desde VS, elegiremos la plantilla “Servicio de Windows”

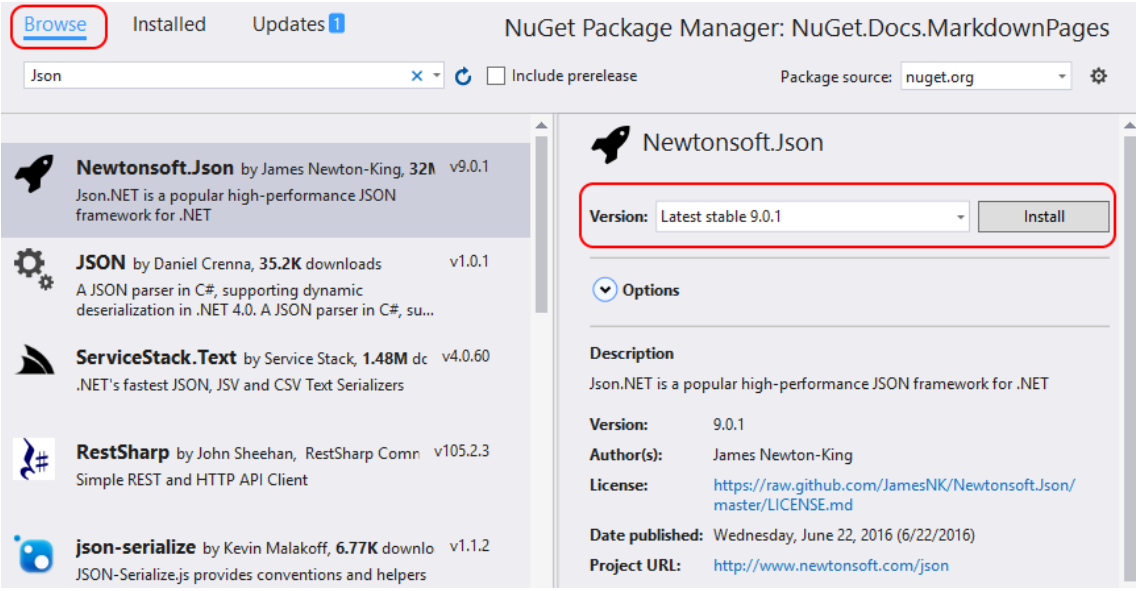


2.3. Administración de paquetes NuGet

La interfaz de usuario del Administrador de paquetes NuGet en Visual Studio de Windows le permite instalar, desinstalar y actualizar fácilmente paquetes NuGet en proyectos y soluciones.

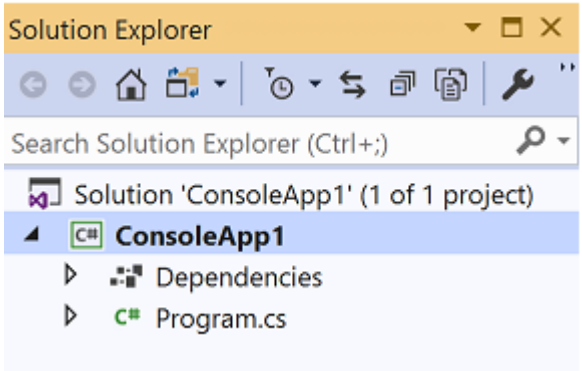
Para acceder a dicha herramienta:

En el Explorador de soluciones, haga clic con el botón derecho en Referencias o en un proyecto y seleccione Administrar paquetes NuGet… .



2.4. Explorador de soluciones

En el Explorador de soluciones, que normalmente se encuentra en el lado derecho de Visual Studio, se muestra una representación gráfica de la jerarquía de los archivos y las carpetas del proyecto, la solución o la carpeta de código. Usando el explorador de soluciones podremos acceder a cualquier elemento del proyecto.

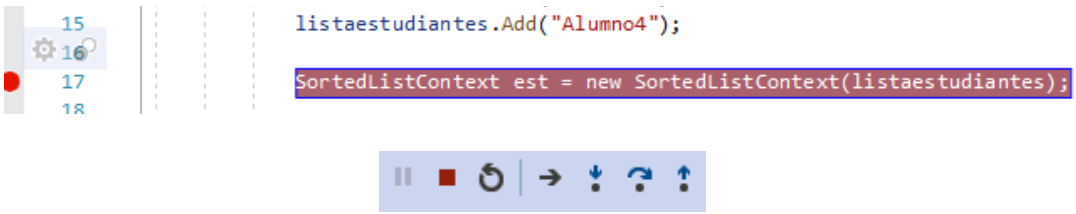


2.5. Debugs y Breakpoints

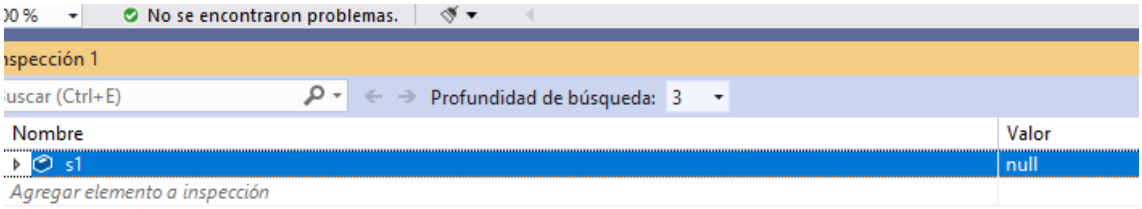
Para iniciar el proceso de depuración, presionamos F5 (Depurar > Iniciar depuración) o el botón Iniciar depuración en la barra de herramientas de depuración. Esto ejecuta el proceso de depuración hasta el primer punto de ruptura de encuentre.

Los puntos de interrupción son una característica de utilidad cuando se conoce la línea o la sección de código que se quiere examinar en detalle.

Para crear un punto de ruptura, se hace click sobre el margen izquierdo de la línea deseada, apareciendo un punto rojo en el momento del click. Esto provocará la parada de la ejecución en dicho punto, permitiéndonos realizar una depuración y estudio más detallado gracias a la barra de depuración.



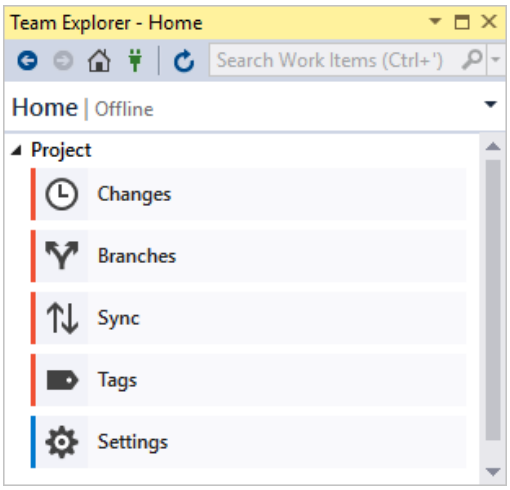
Durante la depuración, también podemos inspeccionar el valor de las variables o expresiones de nuestro código a través de la inspección de variables:

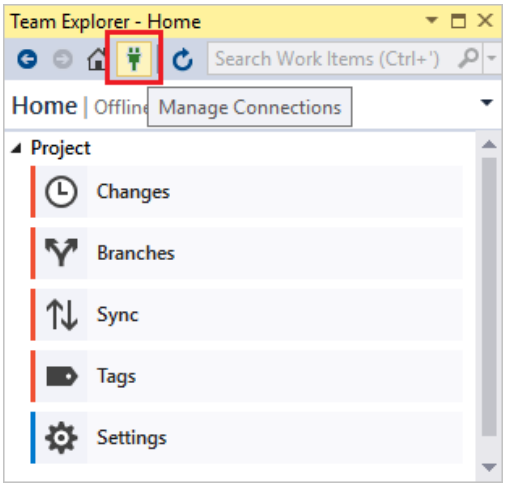


2.6. Team Explorer

En la ventana de la herramienta Team Explorer se coordinarán las versiones de código con otros miembros del equipo para desarrollar un proyecto y para administrar el trabajo que se tenga asignado.

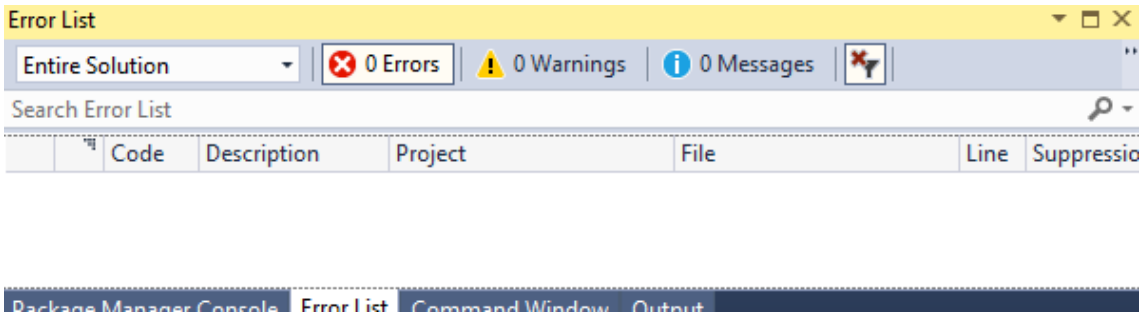
En definitiva, Team Explorer conecta Visual Studio con repositorios de Git y GitHub, repositorios de Control de versiones de Team Foundation (TFVC) y proyectos hospedados en Azure DevOps Services o en una instancia local de Azure DevOps Server (conocido anteriormente como TFS).





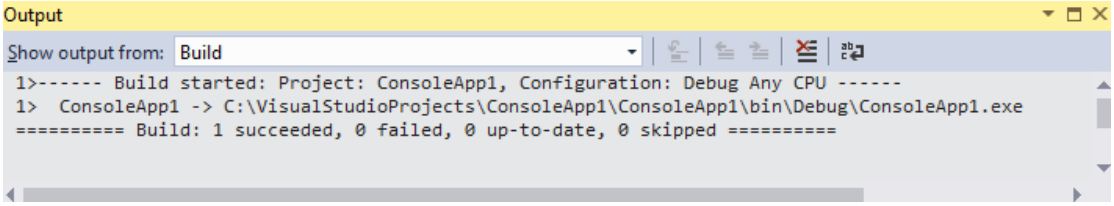
2.7. Lista de Errores

En la Lista de errores se muestran los errores, las advertencias y los mensajes relacionados con el estado actual del código. Si se produjera algún error (por ejemplo, la falta de un paréntesis o un punto y coma) en el archivo o en cualquier parte del proyecto, se indicaría aquí.



2.8. Ventana de Resultados

En la ventana Resultados se muestran los mensajes de resultados de la compilación del proyecto y del proveedor de control de código fuente.



3. POO

3.1. Introducción a la POO

La programación orientada a objetos es un paradigma de programación que utiliza objetos para modelar cosas o ideas del mundo real. Estos objetos se definen mediante las clases. POO utiliza una serie de conceptos y técnicas que, aplicados a estas clases, intenta dar solución a problemas.

El diseño orientado a objetos nos permite utilizar las características de un lenguaje de programación orientado a objetos (conceptos, técnicas y principios) para hacer nuestro software más efectivo y mantenible.

3.1.1. Objetos y Clases

- Un objeto es una instancia de una clase,
- Una clase es una plantilla para la creación de objetos. Es decir, una clase nos sirve para definir un tiempo de datos personalizado, definiendo tanto atributos como comportamiento

Clase

Una clase se compone de atributos y métodos.

Los atributos son los campos y las propiedades de la clase, que sirven para definir los datos de la clase. Los métodos son la funcionalidad o acciones definidas por la clase que aportarán el comportamiento a la misma.

Objeto

Como se comenta anteriormente, un objeto es una instancia de una clase, que se crea para poder utilizar los métodos y propiedades de una clase

Definición de una clase

```
class Clase1
{
    0 referencias
    public int propiedad1 { get; set; }
    0 referencias
    public int propiedad2 { get; set; }
    int campo1;

    0 referencias
    public Clase1(){
    }
    0 referencias
    public void Método1()
    {
    }

}
```

Atributos

Métodos

Definición de un objeto de la clase

```
Clase1 objetoclase = new Clase1();
objetoclase.Método1();
```

A nivel de ejemplo, podríamos comparar los conceptos de clase y objeto con un molde que usamos para crear dulces o magdalenas

3.1.2. Requerimientos funcionales y no funcionales

Cuando escribimos software, lo hacemos basándonos en unos requerimientos que nos proporciona la parte interesada en nuestro desarrollo.

Existen requerimientos Funcionales y requerimientos No Funcionales.

Los requerimientos funcionales son aquellos que describen lo que el software debe hacer.

Los requerimientos No Funcionales son aquellos que hablan de las condiciones en las que se ejecuta nuestro software, como son: Rendimiento, probabilidad, reusabilidad, flexibilidad y mantenibilidad.

3.2. Fundamentos de C# y POO

3.2.1. Campos y Propiedades

Los campos y propiedades representan los atributos de la clase. Se aconseja siempre trabajar con propiedades autoimplementadas.

```
3 referencias
class Clase1
{
    0 referencias
    public string Propiedad1 { get; set; }

    1 referencia
    public Clase1(){
    }
}
```

3.2.2. Constructores

El constructor es una función que se ejecuta al momento de instanciar una clase. Su función principal es la de inicializar el objeto creado.

Los constructores se crean usando el mismo nombre de la clase como nombre del método.

Es posible tener varios constructores de manera simultanea

```
4 referencias
class Clase1
{
    0 referencias
    public string Propiedad1 { get; set; }

    1 referencia
    public Clase1(){
    }

    0 referencias
    public Clase1(string vlaor1, string valor2)
    {
    }
}
```

3.2.3. Signaturas y Sobrecargas

Definimos Signatura al conjunto de nombre y lista de parámetros de un método.

Cuando tenemos dos métodos con el mismo nombre pero diferente listado de parámetros de entrada, se dice que el método está sobrecargado.

Las sobrecargas son útiles cuando tenemos métodos que ejecutan una acción similar pero que necesitan diferente información de entrada

```
1 referencia
public Clase1(){
}

0 referencias
public Clase1(string vlaor1, string valor2)
{
}

0 referencias
public void Metodo1()
{

}
0 referencias
public void Metodo1(string p1)
{

}
0 referencias
public void Metodo1(string p1, int elemento)
{

}
```

3.2.4. Interfaces

Una interfaz son un mecanismo que nos permite definir una serie de signaturas de métodos las cuales, la clase que extienda dicha interfaz se debe comprometer a implementar.

Dicho de otra manera, una interfaz es como un contrato, el cual cada clase que la implemente debe cumplir.

En C#, Un ejemplo de definición de interfaz puede ser el siguiente:

```
public interface IRepositorioPersonas
{
    3 referencias
    void obtenerPersonas();
}
```

Para implementar en C# una clase que extendiera de una interfaz, se haría de la siguiente manera:

```
2 referencias
public class RepositorioPersonasMemoria : IRepositorioPersonas
{
    3 referencias
```

Es posible, extender de varias interfaces a la vez, separándolas por comas después de los dos puntos.

El uso de Interfaces hace que nuestros aplicativos sean mucho más flexibles y versátiles, ya que unificar/enrutamos las definición de las funciones o métodos en una abstracción que podrá tener diferentes implementaciones en función de la clase usada (este término se define polimorfismo).

3.2.5. Clases Estáticas

Cuando se declara algo como estático, quiere decir que su funcionamiento no está relacionado a una instancia de una clase:

- Un método estático es aquel que puede ser usado sin necesidad de instanciar una clase
- Una clase estática es aquella que no pueden ser instanciada. Además, todos sus miembros deben ser estáticos

Las clases estáticas, al no depender de una instancia de la clase, pueden ser usados por ejemplo para funcionalidades genéricas, por ejemplo, creando una clase Utilidades, con métodos genéricos usados "across" en la aplicación.

Ejemplo de definición de clase estática en C#:

```
1 referencia
public static class Utilidades
{
    1 referencia
    public static int calcularPorcentaje(int porc, int total)
    {
        return (int)((total * porc) / 100);
    }
}
```

Ejemplo de uso de clase estática en C#:

```
Console.WriteLine(Utilidades.calcularPorcentaje(50, 100));
```

Hay que tener en cuenta que las clases estáticas no permiten el uso de Interfaces, con lo que ello conlleva, por ejemplo, no poder usar el uso de dependencias, aumentando el acoplamiento. Por este motivo, hay que analizar bien los casos en los que queramos usar una clase estática.

Lo ideal, sería usarlas en casos en los que la funcionalidad sea invariable, universal y que no sea fundamental para nuestro software

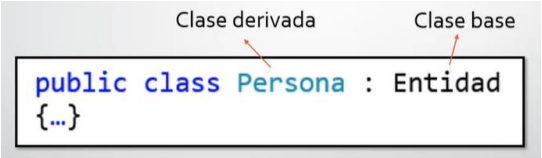
3.3. Herencia

3.3.1. Introducción a Herencia

La herencia es un mecanismo a través del cual podemos construir una clase a partir de otra, compartiendo y reutilizando así código entre clases.

En una relación de herencia, tenemos una clase base (solo una) y una (o varias) clase derivada

En C#, definiremos la herencia de la siguiente manera:



Todas las clases que hemos utilizado heredan de la clase base Object, por lo cual todas las clases que hemos usado y usaremos comparten código con la clase Object.

AL ejecutar una herencia, podremos indicar qué constructor de la clase base queremos ejecutar utilizando ":base", de la siguiente manera (C#):

```
1 referencia
public Perro(string param1):base(param1)
{
```

Clase Abstracta:

Una clase abstracta es una clase que no puede ser instanciada, y su propósito es servir de clase base para otras clases

En C#, definiremos las clases abstractas de la siguiente manera:

```
public abstract class Animal
{...}
```

Sólo en las clases abstractas, podemos definir métodos abstractos. Los métodos abstractos son aquellos que no tienen implementación, la cual debe hacerse en las clases derivadas.

En C#, definimos los métodos abstractos de la siguiente manera:

```
public abstract void HacerRuido();
```

Método virtual:

Un método virtual es aquel que puede ser sobrescrito por una clase derivada y que tiene su implementación.

Para sobrescribir un método virtual, lo hacemos igual que como hicimos con los métodos abstractos, utilizando “override”.

En C#, definiremos los métodos virtuales de la siguiente manera:

```
0 referencias
public virtual void hacerRuido()
{
    Console.WriteLine("Hacer ruido");
}
```

Y lo sobreescribiremos en la clase derivada de la siguiente manera:

```
2 referencias
public override void hacerRuido() //<-sobreescribe el método virtual de la clase base
{
    Console.WriteLine("ruido de perro");
}
```

3.4. Modificadores de Acceso

Los modificadores de acceso nos permiten configurar el nivel de accesibilidad de tipos y miembros.

Public:

El método, miembro o clase definido como public podrá ser accedido desde cualquier sitio, ya sea desde dentro o fuera de la clase, o desde dentro o fuera del proyecto.

Para definir clases, variables, métodos o atributos públicos usaremos la palabra reservada “public”.

Private:

El método, miembro o clase definido como private sólo podrá ser accedido desde dentro de la clase donde se encuentra.

Para definir clases, variables, métodos o atributos privados usaremos la palabra reservada “private”.

Internal:

El método, miembro o clase definido como internal sólo podrá ser accedido desde dentro del proyecto donde se encuentra.

Para definir clases, variables, métodos o atributos privados usaremos la palabra reservada “internal”.

Protected:

El método, miembro o clase definido como protected sólo podrá ser accedido desde dentro de la clase o desde una clase derivada.

Para definir clases, variables, métodos o atributos protected usaremos la palabra reservada “protected”.

Protected Internal:

El método, miembro o clase definido como protected internal sólo podrá ser accedido desde dentro de la clase o desde fuera del proyecto a través de una clase derivada.

Para definir clases, variables, métodos o atributos protected internal usaremos la palabra reservada “protected internal”.

Private Protected:

El método, miembro o clase definido como private protected sólo podrá ser accedido desde dentro de la clase o desde otra clase derivada pero que se encuentre en el mismo proyecto.

Para definir clases, variables, métodos o atributos private protected usaremos la palabra reservada “private protected”.

3.5. Los 4 pilares de la Programación Orientada a Objetos

Existen 4 pilares básicos y fundamentales de la POO: Herencia (ya visto en el punto anterior), Abstracción, Encapsulamiento y Polimorfismo.

Abstracción

La abstracción es una técnica mediante la cual ignoramos las cosas que no son relevantes y nos centramos en los aspectos que realmente aportan valor en nuestro contexto. Para conseguir esto, una de las cosas que debemos plantear es preguntarnos ¿Qué necesitamos para cumplir con los objetivos de la aplicación?

Debemos tener muy en cuenta que añadir partes, características y atributos que no necesitamos en nuestra aplicación complicará nuestro software innecesariamente.

Encapsulamiento

El encapsulamiento es una técnica que consiste en esconder los detalles de implementación de una clase, tratandola básicamente como una caja negra. Para esto usamos los modificadores de acceso estudiados en el tema anterior.

Con esta técnica, además de aumentar la mantenibilidad de nuestro software y simplificar las relaciones, facilita la evolucionabilidad de una clase (gracias a sus elementos privados) y garantiza la integridad del data de la clase.

Finalmente, diremos que además nos da la posibilidad de cambiar la implementación de los elementos no visibles de la clase sin afectar la relación con las otras clases

Polimorfismo

El polimorfismo se refiere a la idea de que varios objetos puedan compartir signaturas de miembros y que, a través de estas signaturas de miembros podamos llamar a los miembros del objeto en cuestión

Existen dos categorías de polimorfismo:

- Polimorfismo por herencia: Distintas clases derivadas pueden sobrescribir métodos abstractos o virtuales de la clase base y al ser llamados, van a ser invocados los de la clase derivada correspondiente.
- Polimorfismo por interfaz: En este caso usamos una interfaz en lugar de una clase base

3.6. Complejidad de clases

Cohesión:

La cohesión es el nivel de coherencia entre los miembros de una clase o la coherencia existente entre las distintas clases de un módulo.

Hablaremos de cohesión alta cuando exista una concordancia o relación lógica entre los elementos existentes en la clase o módulo. Por el contrario, hablaremos de cohesión baja cuando no exista relación entre los elementos de una clase o módulo.

Por otro lado, como hemos adelantado ya, la cohesión se puede establecer también a nivel de módulo, refiriéndonos en este caso a la manera en la que organizamos las clases, interfaces, enums, etc..

Por ejemplo, a continuación un ejemplo de cohesión baja (a la izquierda) y cohesión alta (a la derecha) al hablar de módulos:

Acoplamiento:

El acoplamiento se refiere a la complejidad de las relaciones que existen entre las clases.

Cuando tenemos un sistema estrechamente acoplado (o altamente acoplado), queremos decir que la dependencia entre las clases es compleja.

Cuando tenemos un sistema con acoplamiento débil, quiere decir que la dependencia es simple

Otra técnica que podemos utilizar para garantizar la facilidad del reemplazo de implementaciónes en una relación es la inyección de dependencias.

Veremos en capítulos posteriores de qué se trata la inyección de dependencias, pero diremos que atiende al concepto de que una clase reciba por parámetro las implementaciones de las dependencias que necesita para funcionar.

3.7. Principios SOLID

Solid es un acrónimo inventado por Robert C.Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo. Los costes de mantenimiento pueden abarcar el 80% de un proyecto de software por lo que hay que valorar un buen diseño.

3.7.1. Principio de responsabilidad única

El principio de responsabilidad única nos dice que cada módulo de un software debe tener una única razón para cambiar. Entendemos como responsabilidad como un objetivo o fin que queremos alcanzar dentro de un módulo.

Este principio está relacionado con el concepto de cohesión. Cuando tenemos una alta cohesión, los elementos del módulo se relacionarán entre sí de una manera lógica; cuando

tenemos una baja cohesión, los elementos del módulo tendrán una baja o nula relación entre sí, y por tanto varias responsabilidades.

Al encontrarnos con un código que no cumple el principio de responsabilidad única, debemos plantearnos si debe ser modificado o no, en función de si el software va a ser mantenido en un futuro.

Aplicando el principio de responsabilidad única, conseguimos:

- Reutilización de código
- Cambio de implementación sin afectación a otros módulos
- Las pruebas serán más fáciles de escribir

3.7.2. Principio de abierto/cerrado

El principio de abierto/cerrado nos dice que los módulos deben ser abiertos para ser extendidos y cerrados para ser modificados.

Esto quiere decir que nuestro software debe ser fácilmente extensible a través de sus dependencias y tan unitario en su responsabilidad, que quede ajeno a cualquier modificación que no tenga que ver con su única función.

3.7.3. Principio de sustitución de Liskov

Este principio, que se basa en el concepto de polimorfismo, nos dice que si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de sus clases hijas y que el programa siga siendo válido.

Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de clase padre padre.

Ejemplo 1:

Creamos una clase que extiende de otra, pero de repente uno de los métodos nos sobra, y no sabemos qué hacer con él.

Las opciones más rápidas son bien dejarlo vacío o bien lanzar una excepción cuando se use, asegurándote de que nadie llama incorrectamente a un método que no se puede utilizar. Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estemos violando el principio de sustitución de Liskov, ya que si nuestro código estaba usando un método que para algunas concreciones lanzamos una excepción, ¿cómo podemos estar seguros de que todo sigua funcionando?

Ejemplo 2:

Definimos una clase para calcular la superficie de un rectángulo, pero la definimos tan concretamente, que si extendemos de ella para obtener el área de un cuadrado no nos vale:

```
public class Rectangle
{

    private int width;
    private int height;
    0 referencias
    public int getWidth()
    {
        return width;
    }
    0 referencias
    public void setWidth(int width)
    {
        this.width = width;
    }
    0 referencias
    public int getHeight()
    {
        return height;
    }
    0 referencias
    public void setHeight(int height)
    {
        this.height = height;
    }
    0 referencias
    public int calculateArea()
    {
        return width * height;
    }
}
```

```
@Test
public void testArea() {
    Rectangle r = new Rectangle();
    r.setWidth(5);
    r.setHeight(4);
    assertEquals(20, r.calculateArea());
}
```

La definición del cuadrado sería la siguiente:

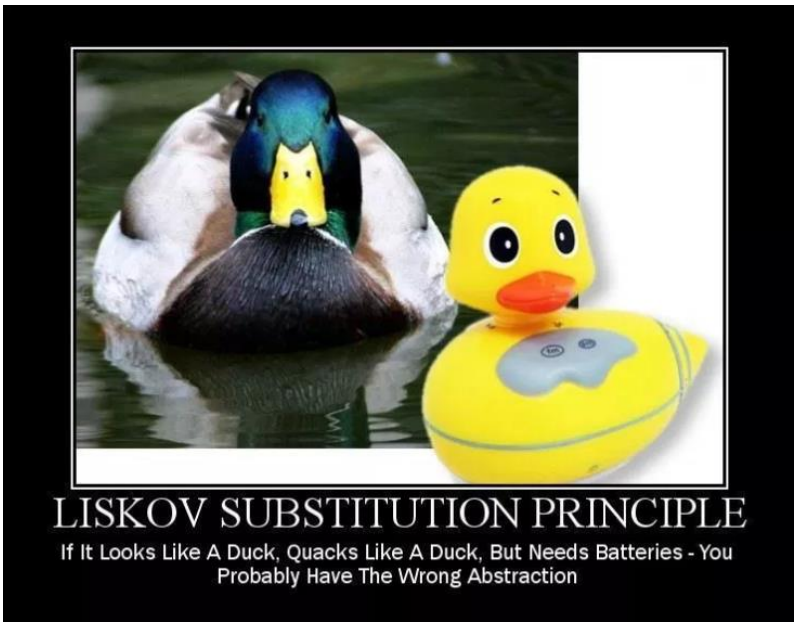
```
public class Square : Rectangle
{
    2 referencias
    public void setWidth(int width)
    {
        setWidth(width);
        setHeight(width);
    }
    2 referencias
    public void setHeight(int height)
    {
        setHeight(height);
        setWidth(height);
    }
}
```

Si ejecutamos el test vemos que no se cumple, el resultado sería 16 en lugar de 20. Estamos por tanto violando el principio de sustitución de Liskov.

Para solucionar este problema tendríamos varias posibilidades en función del caso en el que nos encontremos. Lo más habitual será ampliar esa jerarquía de clases. Podemos extraer a otra clase padre las características comunes y hacer que la antigua clase padre y su hija hereden de ella. Al final lo más probable es que la clase tenga tan poco código que acabemos teniendo una simple interfaz:

```
public interface IRectangle
{
    1 referencia
    int getWidth();
    1 referencia
    int getHeight();
    1 referencia
    int calculateArea();
}
2 referencias
public class Rectangle : IRectangle
{}
1 referencia
public class Square : IRectangle
{}
```

Ejemplo 3:



3.7.4. Principio de segregación de interfaces

El principio de segregación de interfaces nos dice que nuestras interfaces sólo deberían tener las operaciones necesarias para cumplir un conjunto de tareas relacionadas. Es decir, nuestras interfaces deben ser cohesivas.

Los clientes no deberían estar obligados a usar métodos que no van a usar.

Ejemplo:

Imaginemos que tenemos una tienda de venta de CDs. Para dicha tienda, hemos creado una interfaz denominada Producto la cual implementa el producto CD de la siguiente manera:

```
public interface Product
{
    1 referencia
    String getName();
    1 referencia
    int getStock();
    1 referencia
    int getNumberOfDisks();
}
```

Si posteriormente, empezamos a vender películas en DVD y decidimos usar la misma Interfaz, quizás tengamos que añadir métodos que no aplican a los CDs, como por ejemplo, la edad recomendada de la película del DVD.

Esto puede provocar que la clase CD implemente el método getAgeRecommmended() sin aplicarle.

3.7.5. Principio de inversión de dependencias

El principio de inversión de dependencias es una técnica básica, muy usada en el día a día y que permitirá que nuestro código sea flexible y mantenible. Consiste en hacer que el código que es el núcleo de nuestra aplicación no dependa de los detalles de implementación, como pueden ser el framework que se use, la base de datos, cómo te conectemos al servidor, etc.

Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

Dicho de otra manera:

Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Ejemplo:

A continuación, un software hecho para realizar un pago en una tienda en el que no se aplica la inversión de dependencias ya que vemos que un método de alto nivel está dependiendo directamente de un método de bajo nivel como es el guardar en base de datos. No existe forma de incluir modificaciones en este código sin desmontar lo que tenemos implementado en este momento.

```
0 referencias
public class ShoppingBasket
{
    0 referencias
    public void buy(Shopping shopping)
    {
        SqlDatabase db = new SqlDatabase();
        db.guardar(shopping);

        CreditCard creditCard = new CreditCard();
        creditCard.pagar(shopping);
    }
}
2 referencias
public class SqlDatabase
{
    1 referencia
    public void guardar(Shopping shopping)
    {
        // instrucciones de guardado
    }
}
2 referencias
public class CreditCard
{
    1 referencia
    public void pagar(Shopping shopping)
    {
        // formalizar el pago
    }
}
```

Para solucionar el problema, primero debemos de dejar de depender de concreciones. Vamos a crear interfaces que definan el comportamiento que debe dar una clase para poder funcionar como mecanismo de persistencia o como método de pago.

```
public interface Persistence
{
    2 referencias
    void guardar(Shopping shopping);
}
2 referencias
public class SqlDatabase : Persistence
{

    2 referencias
    public void guardar(Shopping shopping)
    {
        // Guardar datos en BBDD
    }
}

1 referencia
public interface PaymentMethod
{
    0 referencias
    void pay(Shopping shopping);
}
2 referencias
public class CreditCard : PaymentMethod
{

    1 referencia
    public void pagar(Shopping shopping)
    {
        // Pagar usando tarjeta de credito
    }
}
```

Ahora ya no dependemos de la implementación particular que decidamos. Pero aún tenemos que seguir instanciándolo en el cuerpo.

Nuestro último paso es invertir las dependencias. Vamos a hacer que estos objetos se pasen por constructor

```
private Persistence persistence;
private PaymentMethod paymentMethod;
0 referencias
public ShoppingBasket(Persistence persistence, PaymentMethod paymentMethod)
    {
        this.persistence = persistence;
        this.paymentMethod = paymentMethod;
    }
    0 referencias
    public void buy(Shopping shopping)
    {
        persistence.guardar(shopping);
        paymentMethod.pagar(shopping);
    }
}
```

4. Fundamentos de UML

4.1. Introducción a UML

El lenguaje de modelado unificado (UML) es un estándar para la representación visual de objetos, estados y procesos dentro de un sistema. Por un lado, el lenguaje de modelado puede servir de modelo para un proyecto y garantizar así una arquitectura de información estructurada; por el otro, ayuda a los desarrolladores a presentar la descripción del sistema de una manera que sea comprensible para quienes están fuera del campo. UML se utiliza principalmente en el desarrollo de software orientado a objetos. Al ampliar el estándar en la versión 2.0, también es adecuado para visualizar procesos empresariales.

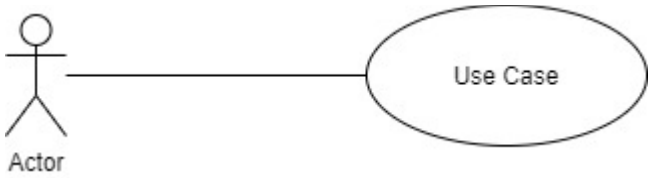
El Lenguaje Unificado de Modelado es referido por algunos como la lengua franca entre los lenguajes de modelado. UML visualiza los estados y las interacciones entre objetos dentro de un sistema. Su extensa popularidad se debe probablemente a la fuerte influencia que ejercen los miembros del OMG (IBM, Microsoft y HP entre otros). La semántica estructurada hace el resto. Los diagramas UML se utilizan para representar los siguientes componentes del sistema:

- Objetos individuales (elementos básicos)
- Clases (combina elementos con las mismas propiedades)
- Relaciones entre objetos (jerarquía y comportamiento/comunicación entre objetos)
- Actividad (combinación compleja de acciones/módulos de comportamiento)
- Interacciones entre objetos e interfaces

4.2. Casos de uso

La técnica de caso de uso captura información sobre el funcionamiento actual o deseado de un sistema o negocio. Aunque no es un verdadero procedimiento orientado a objetos, la técnica de caso de uso le ayuda a construir escenarios que modelan los procesos del sistema. Es una forma excelente de entrar en el análisis de sistemas orientado a objetos.

Normalmente, los diagramas de Caso de uso se modelan para cada escenario en el sistema o negocio. Cada caso de uso se puede definir con el texto que describe el escenario. También puede definir el escenario con la secuencia de pasos realizados en él o con las condiciones que existen antes de que el escenario comience o después de que se haya completado.

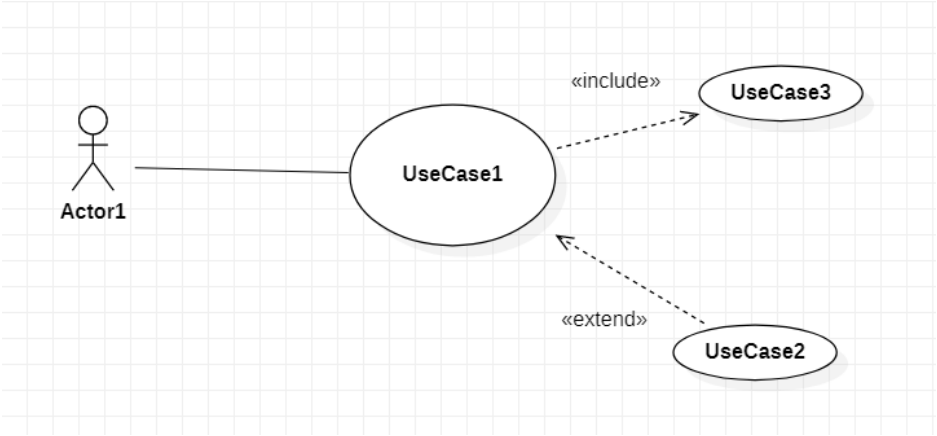


Se conoce como actor a un agente externo a un sistema: alguien o algo que solicita un servicio del sistema o que actúa como catalizador para que suceda algo en el sistema. UML especifica que el actor es una clase de objetos, no una instancia de una clase (u objeto). El actor está representado como una clase con un estereotipo de actor.

Subescenarios:

Se identifican procedimiento adicionales o tareas para hacer posible el normal desarrollo del Caso de uso implicado. Esto es algo así, como los requisitos o políticas internas de la organización, verificaciones, autorizaciones, revisiones, etc, para poder realizar la principal actividad del Caso de uso. A estos elementos se les denomina “Subescenarios”.

Los subescenarios, pueden ser incluídos o extendidos. Se usan como “include”`s cuando el Subescenario es consumido/usado/implicado para el caso de uso principal. Se usan como “extend”’s cuando son conseciencias/resultado del uso de otro subescenario a nivel “include”.

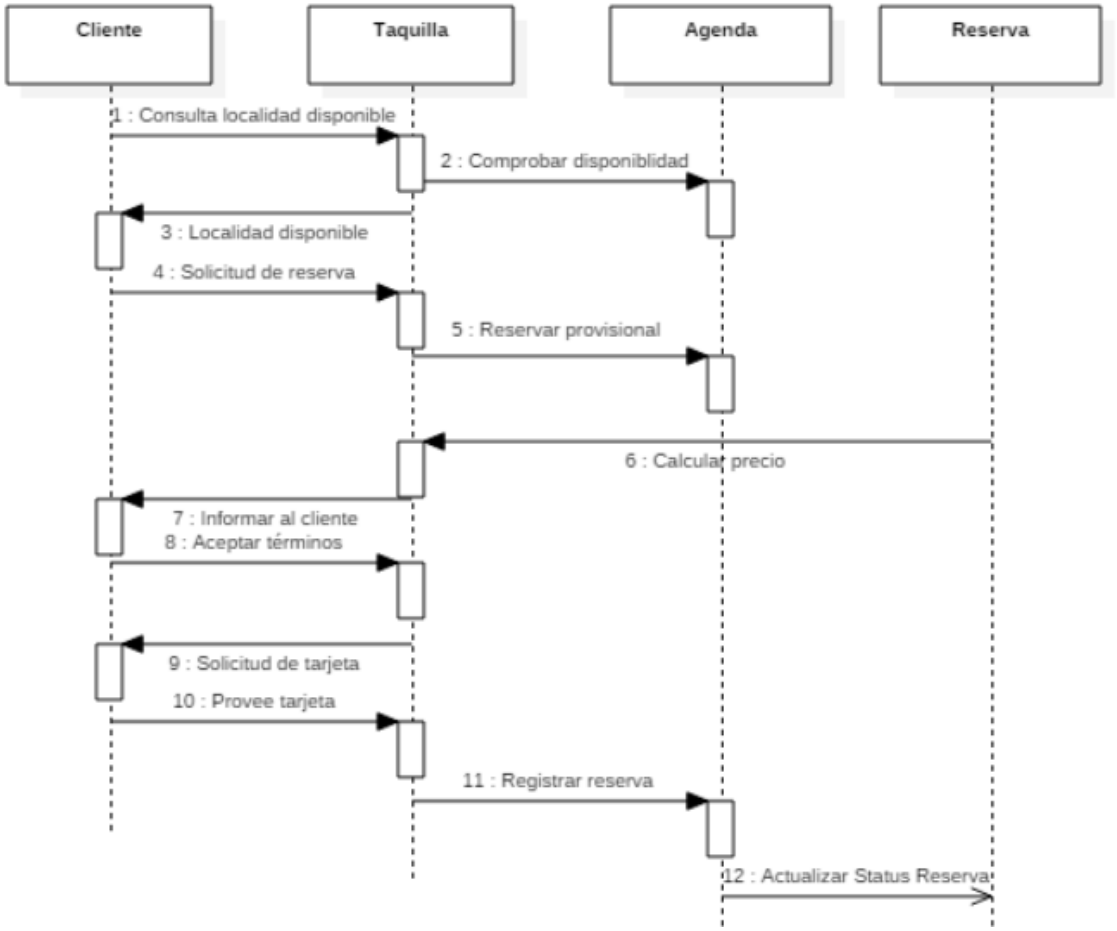


4.3. Diagramas

4.3.1. Diagrama de secuencia

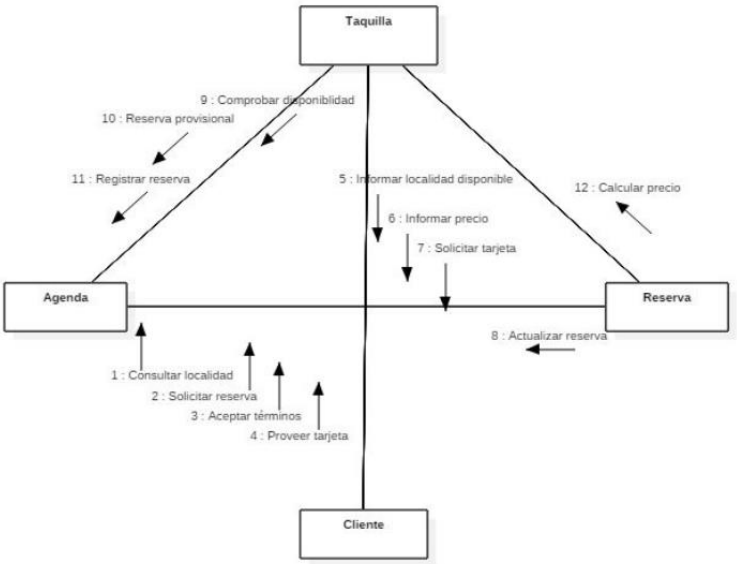
El diagrama de secuencia ofrece una vista de **bajo nivel** que contiene detalles de implementación del escenario, incluyendo los objetos, las clases utilizadas y los mensajes pasados entre los objetos.

En el diagrama de secuencia, se utilizan líneas verticales para representar objetos y vectores horizontales para representar los mensajes pasados.



4.3.2. Diagrama de comunicación/colaboración

El diagrama de comunicación/colaboración, basándose en el diagrama de secuencia , permite trasladar los mensajes entre objetos y convertirlos a actividades, funciones y propiedades para el diseño final del software.



4.3.3. Diagrama de clases

El diagrama de clase describe la identidad del sistema, su relación con otras clases, sus atributos y sus operaciones.

El diagrama de Clase se compone de los siguientes elementos::

Asociación entre clases. Se representa gráficamente mediante una línea sólida que conecta una clase con otra. Dicha asociación tendrá un nombre, el cual debe describir la tarea general que relaciona los objetos o clases

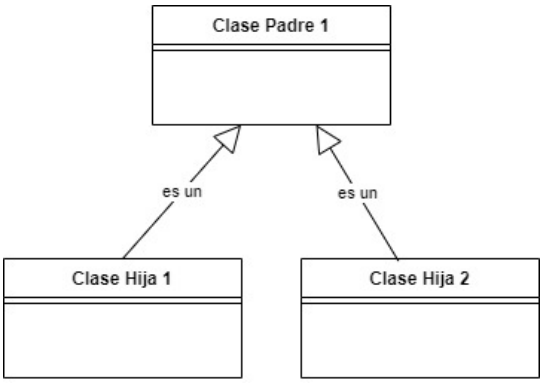
Debe definirse la multiplicidad de la asociación:

Multiplicidad	Descripción
0..1	Cero o una instancia
1	Exactamente 1 instancia
0..*	Cero o varias instancias
1..*	Una o varias instancias
n	n instancias

Se definirán herencias y referencias cruzadas de diagramas de secuencia y clase.

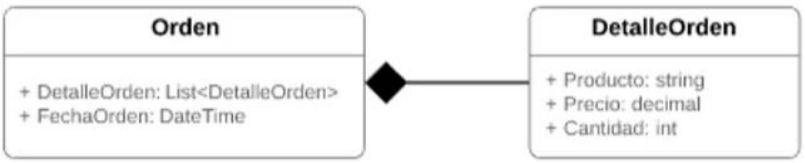
Herencia:

Podemos identificar la herencia en un diagrama UML a través de la siguiente forma:



Composición:

La relación de composición (tiene-un) en un diagrama UML se define gráficamente de la siguiente forma:



Atributos y Operaciones:

Se observa tres espacios en la representación de la clase, el primero espacio alberga el id o nombre de la clase, el segundo espacio será para los atributos de la clase y el tercer espacio almacenará los métodos de cada clase.

Los atributos en un modelo de Clases, corresponden a las características o rasgos propios de un objeto.

Por ejemplo:

- Peso, raza, estatura, edad: son atributos candidatos para una clase persona
- Velocidad, número de sillas, tipo de gasolina: son propios de una clase vehículo

La operaciones determinan el comportamiento de cada objeto en el sistema

Por ejemplo:

- Un objeto Avión, debe realizar ciertas tareas para hacerlo operable: Despegar, Aterrizar, Descender, etc
- El objeto calculadora digital debe Sumar, Sacar raíz cuadrada, calcular el coseno de un ángulo, etc.

En la siguiente tabla, se identifican los modificadores de acceso aplicables a los elementos de la clase:

Símbolo	Modificador de Acceso
+	public
-	private
#	protected
~	internal

4.3.4. Diagrama de estado

Este tipo de diagrama modela la secuencia de estados por los que pasa un objeto de una clase durante su vida. El diagrama de estado capta los estímulos recibidos por el objeto, las respuestas y las acciones.

El estado de los objetos puede encontrarse durante la vida del sistema, en cualquiera de estas situaciones:

- Determinado por los Atributos. Los datos que se relacionan con el objeto, determinan el estado actual. Son los estados de sus atributos

Ejemplo: Un estudiante está en estado de “Soltero” durante su infancia. En su edad adulta podrá estar “Casado”. Este estado de la persona en la línea del tiempo podría llamarse “Estado civil”.

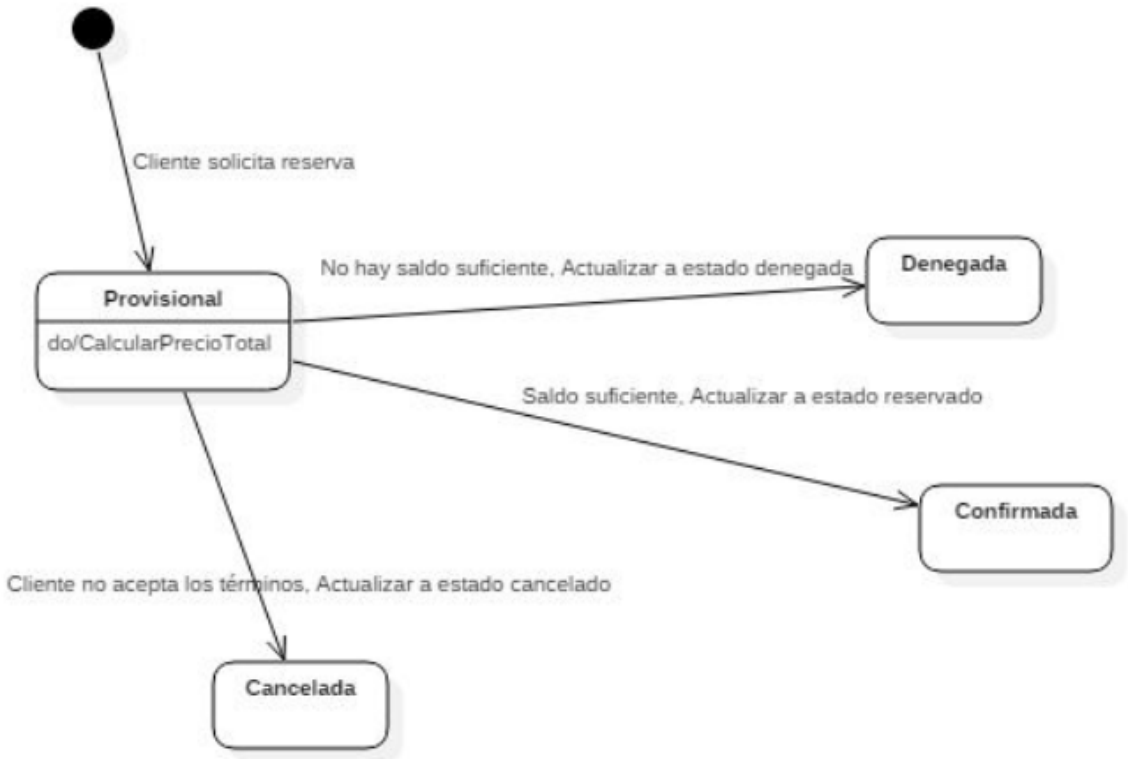
- Determinado por las Acciones de un Objeto. Las acciones que ejecuta el objeto en un momento determinado.

Ejemplo: Un auto inicia su recorrido. El objeto se halla en estado “Transitando”

- Pasivo o En Espera. Generalmente los Objetos se hallan en Stand By o a la espera de ser llamados o utilizados. Aunque están creados en memoria, no necesariamente se encuentran en acción.

Ejemplo de diagrama de estado:

Observe que el objeto "Reserva" envía un mensaje "calcular precio" al momento de generar la reserva provisional. Esto es un método que genera la definición de transición "Saldo suficiente", y se ejecuta cuando el objeto se halla en el estado "Provisional", pues solo el cliente reservara definitivamente cuando conozca el precio final, para que la reserva pase al estado "Confirmada"



5. Arquitectura de aplicaciones

5.1. Arquitectura de N-Capas

La arquitectura basada en N-Capas se enfoca en la distribución de una aplicación, en diferentes niveles o capas, proveyendo de esta forma una efectivo desacoplamiento entre los componentes del aplicativo.

El modelo de arquitectura de N-Capas mayormente extendido es el de 3 capas, en el cual poseemos capa presentación, capa de negocio y capa de datos

Características de la arquitectura N-Capas:

*Los componentes del aplicativo, se organizan en capas diferenciadas que se comunican entre sí.

*Cada capa tiene un rol y una responsabilidad única dentro de la aplicación: Cada capa debe estar pensada para cumplir un único propósito. Esta idea o característica, está muy relacionada con los principios de diseño SOLID, en concreto con el primer principio de "Única Responsabilidad"

*Existen capas de alto nivel y de bajo nivel. Las capas de alto nivel están más cerca de la interacción con el usuario, y las capas de más bajo nivel están más cerca de acciones más relacionadas con el Core. Por este motivo, existirán tareas organizadas en alto nivel y bajo nivel.

Recomendaciones del diseño de N-Capas:

*Una capa superior, tiene conocimiento de la capa inmediatamente inferior pero no al revés, lo cual asegura la correcta organización de la dependencia entre dichas capas.

*Cada capa crea una abstracción de la funcionalidad de ofrece. Dicho de otro modo: una capa superior consumirá un determinado método de una capa inmediatamente inferior sin importarle el “cómo” se hace, sino el “qué” hace.

Ventajas de la arquitectura N-Capas:

*El desarrollo se puede llevar a cabo en varios niveles y, en caso de que sobrevenga algún cambio, solo afectará al nivel requerido sin tener que revisar el código fuente de otros módulos.

*Permite distribuir el trabajo de creación de una aplicación por niveles; de este modo, cada grupo de trabajo está totalmente abstraído del resto de niveles, de forma que basta con conocer la API que existe entre niveles.

*Se mejora la mantenibilidad de la aplicación, facilitando la codificación, reutilización e incluso sustitución de cualquiera de las capas implicadas.

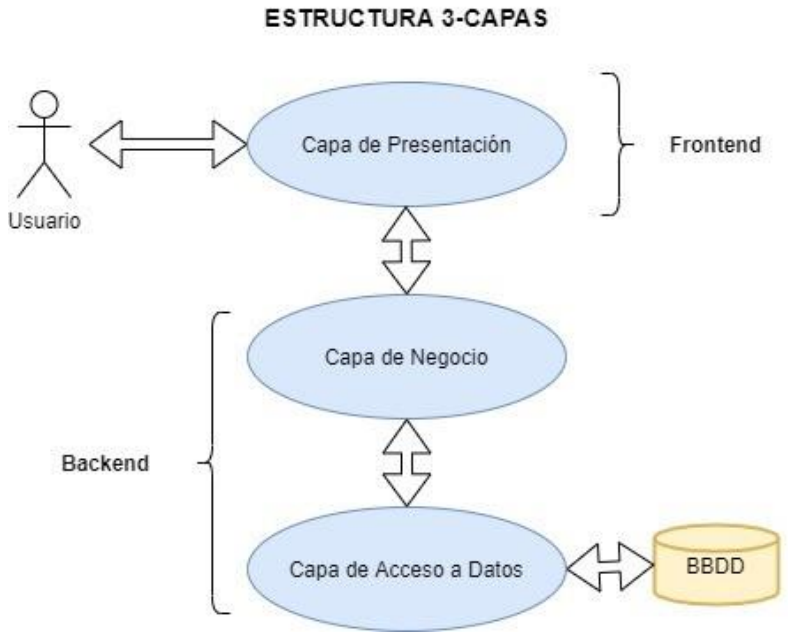
Arquitectura de 3-Capas:

La arquitectura de 3-Capas es el modelo N-Capas usado con mayor frecuencia: Se identifican las siguientes capas:

*Capa presentación: Es la capa que interactúa con las acciones del usuario. Se denomina también Frontend

*Capa de negocio: Formada por las entidades, que representan los objetos que van a ser manejados o utilizados por toda la aplicación. Se implementan en ella todos los procesos de negocio sobre las diferentes entidades. Es el nexo de unión entre presentación y datos.

*Capa de acceso a datos: Contiene las clases que realizan la persistencia de los datos.



5.2. Patrones de diseño

5.2.1. ¿Qué son los 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

Los patrones de diseño nos ayudan a cumplir los principios de programación SOLID. Control de cohesión y acoplamiento o reutilización de código son algunos de los beneficios que podemos conseguir al utilizar patrones, además del ahorro de tiempo, la validación código y el uso de un lenguaje común.

Existen tres tipos de patrones: Patrones Creacionales, Patrones Estructurales y Patrones de Comportamiento, de los cuales, nosotros estudiaremos los siguientes: DAO, Repository, Facade, Factory, Command, Strategy, Singleton

5.2.2. Patrones de diseño creacionales

Estos patrones se centran en la creación de instancias , encapsulando y abstrayendo dicha creación.

Factory Method:

Podemos utilizar este patrón cuando definamos una clase a partir de la que se crearán objetos sin saber de qué tipo son, siendo otras subclases las encargadas de decidirlo.

Este patrón es usado definiendo una especie de “factoria” que nos dará los diferentes objetos en tiempo de ejecución.

Ejemplo de implementación del patrón:

```
5 referencias
public interface IMoto
{
    3 referencias
    public int getRuedas();
}

1 referencia
public interface IFactory
{
    2 referencias
    public IMoto creaMoto(string tipo, int ruedas);
}
```

```
2 referencias
class MotoAgua:IMoto
{
    int ruedas;
    1 referencia
    public MotoAgua(int ruedas)
    {
        this.ruedas = ruedas;
    }

    3 referencias
    public int getRuedas()
    {
        return this.ruedas;
    }
}
2 referencias
class MotoCampo : IMoto
{
    int ruedas;
    1 referencia
    public MotoCampo(int ruedas)
    {
        this.ruedas = ruedas;
    }

    3 referencias
    public int getRuedas()
    {
        return this.ruedas;
    }
}
```

```
2 referencias
public class FactoryIMP:IFactory
{
    public const string agua = "agua";
    public const string campo = "campo";

    2 referencias
    public IMoto creaMoto(string tipo, int ruedas)
    {
        switch (tipo)
        {
            case agua:
                return new MotoAgua(ruedas);
            case campo:
                return new MotoCampo(ruedas);
            default: return null;
        }
    }
}
```

```
0 referencias
static void Main(string[] args)
{
    string tipoMoto = "campo";
    int numRuedas = 2;

    FactoryIMP mifactoria = new FactoryIMP();
    IMoto mimoto = mifactoria.creaMoto(tipoMoto, numRuedas);

    Console.WriteLine(mimoto.getRuedas());
}
```

En el código mostrado, creamos la interfaz IMoto que será la base del objeto a crear y la interfaz Factory que será la encargada de crear encapsuladamente dichos objetos. Teniendo presente que (para nuestro ejemplo concreto), podemos comprarnos una moto de campo o de agua, definimos la clase correspondiente MotoAgua y MotoCampo, que implementan la interfaz IMoto.

Después de esto, creamos la clase Factoria, que implementa IFactoria.

Con dicha estructura montada, codificamos el cuerpo del aplicativo, creando un objeto a partir de nuestra clase factoria y llamando a su método creaMoto.

Singleton:

Singleton es un patrón de diseño del tipo creacional cuyo propósito es garantizar la existencia de una sola instancia de una clase. Además el acceso a esa única instancia tiene que ser global.

Ejemplo de implementación del patrón:

```
8 referencias
class Singleton
{
    private static Singleton _instance = null;
    1 referencia
    private Singleton()
    {
    }

    2 referencias
    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            return _instance;
        }
    }
}
```

```
0 referencias
static void Main(string[] args)
{
    Singleton s1 = Singleton.Instance;
    Singleton s2 = Singleton.Instance;

    if (Object.ReferenceEquals(s1, s2))
    {
        Console.WriteLine("Singleton is working");
    }
    else
    {
        Console.WriteLine("Singleton is broken");
    }
}
```

En el código mostrado, creamos la clase Singleton con un constructor privado para que no pueda ser llamado desde ningún otro sitio.

5.2.3. Patrones de diseño estructurales

Los patrones estructurales describen como usar estructura de datos complejas a partir de otras más simples.

Facade:

El patrón Facade o Fachada viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre estos.

Ejemplo de implementación del patrón:

```
2 referencias
public class Cajero
{
    1 referencia
    public int introducirCantidad() { return 0;  }
    1 referencia
    public bool tieneSaldo(int cantidad) { return false; }
    1 referencia
    public int expedirDinero () { return 0; }
    1 referencia
    public String imprimirTicket() { return "";  }
}
```

```
public class Cuenta
{
    1 referencia
    public double comprobarSaldoDisponible() { return 0; }
    1 referencia
    public bool bloquearCuenta() { return false; }
    1 referencia
    public bool desbloquearCuenta() { return false;  }
    1 referencia
    public void retirarSaldo(int cantidad) { }
    1 referencia
    public bool actualizarCuenta() { return false; }
    1 referencia
    public void alFallar() { }
}
```

```
2 referencias
class Autentificacion
{
    1 referencia
    public bool leerTarjeta() { return false; }
    1 referencia
    public String introducirClave() { return ""; }
    1 referencia
    public bool comprobarClave(String clave) { return false;  }
    1 referencia
    public Cuenta obtenerCuenta() { return null; }
    1 referencia
    public void alFallar() { }
}
```

Creamos las 3 clases básicas para la implementación de operaciones en una cuenta desde un cajero. En este escenario, el cliente tendría que acceder a 3 subsistemas y realizar una inmensa cantidad de operaciones. Sin embargo, crearemos una fachada para ofrecer una interfaz mucho más amigable

```
public class FachadaCajero
{
    private Autentificacion autentificacion = new Autentificacion();
    private Cajero cajero = new Cajero();
    private Cuenta cuenta = null;

    1 referencia
    public void introducirCredenciales()
    {
        bool tarjeta_correcta = autentificacion.leerTarjeta();
        if (tarjeta_correcta)
        {
            String clave = autentificacion.introducirClave();
            bool clave_correcta = autentificacion.comprobarClave(clave);
            if (clave_correcta)
            {
                cuenta = autentificacion.obtenerCuenta();
                return;
            }
        }
        autentificacion.alFallar();
    }
    1 referencia
    public void sacarDinero()
    {
        if (cuenta != null)
        {
            int cantidad = cajero.introducirCantidad();
            bool tiene_dinero = cajero.tieneSaldo(cantidad);
            if (tiene_dinero)
            {
                double hay_saldo_suficiente = cuenta.comprobarSaldoDisponible();
                if (hay_saldo_suficiente>0){
                    cuenta.bloquearCuenta();
                    cuenta.retirarSaldo(cantidad);
                    cuenta.actualizarCuenta();
                    cuenta.desbloquearCuenta();
                    cajero.expedirDinero();
                    cajero.imprimirTicket();
                }else{
                    cuenta.alFallar();
                }
            }
        }
```

Se crea una clase que unifica todas las operaciones de las 3 clases iniciales en una sola. De esta manera, hemos conseguido que para que un cliente use el cajero, no tenga que realizar todas las operaciones de sus subsistemas. En lugar de ello proporcionamos una interfaz mucho más simple que facilita enormemente su uso:

```
0 referencias
static void Main(string[] args)
{
    FachadaCajero cajero_automatico = new FachadaCajero();
    cajero_automatico.introducirCredenciales();
    cajero_automatico.sacarDinero();
}
```

DAO:

Es un patrón de diseño que permite disponer de un componente de actúe como puente de implementación entre los datos almacenados en los objetos de la lógica de negocio y los datos de persistencia.

Encapsula la lógica necesaria para copiar los valores de los datos desde las clases del problema (capa de negocio) hacia la persistencia (capa de datos)

Suele ofrecer métodos para añadir,actualizar, buscar y borrar elementos.

Se compone de:

- BusinessObject: Objeto de la capa de negocio que necesita acceder a la fuente de datos
- DataAccessObject (DAO): Abstrae la capa de negocio de la implementación del acceso a datos.
- DataTRansferObject: (DTO): Objeto contenedor de transferencia de los datos.
- DataSource: fuente de datos;

Ejemplo de Implementación del patrón:

```
1 referencia
class RegistroEmergencia
{
    private string numero;
    private string nombrepaciente;
    private DateTime fecha;

    0 referencias
    public RegistroEmergencia(string numero, string nombrepaciente, DateTime fecha)
    {
        this.numero = numero;
        this.nombrepaciente = nombrepaciente;
        this.fecha = fecha;
    }
}
```

```
7 referencias
public class DTORegistroEmergencia
{
    private string numero;
    private string nombrepaciente;
    private DateTime fecha;

    1 referencia
    public DTORegistroEmergencia(string numero, string nombrepaciente, DateTime fecha)
    {
        this.numero = numero;
        this.nombrepaciente = nombrepaciente;
        this.fecha = fecha;
    }
}
```

```
1 referencia
public interface IDAORegistroEmergencia
{
    1 referencia
    public void CrearRegistroEmergencia (DTORegistroEmergencia registro);
    1 referencia
    public DTORegistroEmergencia buscarRegistroEmergencia(string numero);
}
```

```
0 referencias
public class DAODBImpRegistroEmergencia : IDAORegistroEmergencia
{
    1 referencia
    public void CrearRegistroEmergencia(DTORegistroEmergencia registro)
    {
        /*codificación/reutilizacion de conexión usada*/
    }
    1 referencia
    public DTORegistroEmergencia buscarRegistroEmergencia(string numero)
    {
        /*codificación/reutilizacion de conexión usada*/
        DTORegistroEmergencia registro = new DTORegistroEmergencia(numero, "", new DateTime());
        return registro;
    }
}
```

Repository:

El patron Repository se define como un mecanismo para encapsular el comportamiento de almacenamiento, obtención y búsqueda, de una forma similar a una colección de objetos.

Ejemplo de Implementación del patrón:

```
2 referencias
public class UserRepository
{
    1 referencia
    public User getUserByUsername(String username) {
        User usr = new User();
        return usr;
    }
}
```

```
0 referencias
static void Main(string[] args)
{
    UserRepository userrepo = new UserRepository();
    User u = userrepo.getUserByUsername("juanjose");
}
```

Implementamos la acción de búsqueda de un usuario en una clase que podría implementar además otras acciones relacionadas con el usuario.

5.2.4. Patrones de comportamiento

Especifican las relaciones y acciones entre componentes de nuestro aplicativo.

Command:

Este patrón permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además facilita la parametrización de los métodos.

Ejemplo de implementación del patrón:

```
public class Cuenta
{
    private int id;
    private double saldo;

    1 referencia
    public Cuenta (int id, double saldo)
    {
        this.id = id;
        this.saldo = saldo;
    }

    1 referencia
    public void retirar (double importe)
    {
        this.saldo = this.saldo - importe;
    }
    1 referencia
    public void depositar(double importe)
    {
        this.saldo = this.saldo + importe;
    }
}
```

```
6 referencias
public interface IOperacion
{
    3 referencias
    void execute();
}
```

```
3 referencias
public class DepositarImpl : IOperacion
{
    private Cuenta cuenta;
    private double importe;

    1 referencia
    public DepositarImpl(Cuenta cuenta, double importe)
    {
        this.cuenta = cuenta;
        this.importe = importe;
    }

    3 referencias
    public void execute()
    {
        this.cuenta.depositar(this.importe);
    }
}
```

```
3 referencias
public class RetirarImpl : IOperacion
{
    private Cuenta cuenta;
    private double importe;

    1 referencia
    public RetirarImpl(Cuenta cuenta, double importe)
    {
        this.cuenta = cuenta;
        this.importe = importe;
    }

    3 referencias
    public void execute()
    {
        this.cuenta.retirar(this.importe);
    }
}
```

```
2 referencias
public class Invoker
{
    private List<IOperacion> operaciones = new List<IOperacion>();

    2 referencias
    public void recibirOperacion(IOperacion operacion)
    {
        this.operaciones.Add(operacion);
    }

    1 referencia
    public void realizarOperaciones()
    {
        foreach (IOperacion op in this.operaciones){
            op.execute();
        }
        this.operaciones.Clear();
    }
}
```

```
0 referencias
class Program
{
    0 referencias
    static void Main(string[] args)
    {
        Cuenta cuenta = new Cuenta(1, 200);

        DepositarImpl opDepositar = new DepositarImpl(cuenta, 100);
        RetirarImpl opRetirar = new RetirarImpl(cuenta, 100);

        Invoker ivk = new Invoker();
        ivk.recibirOperacion(opDepositar);
        ivk.recibirOperacion(opDepositar);

        ivk.realizarOperaciones();

    }
}
```

Creamos la clase Cuenta, con las dos operaciones básicas. Una interfaz IOperacion con un único método execute. Creamos dos clases para las posibles operaciones, como son DepositarImpl y RetirarImpl que implementan la interfaz IOperacion.

Por último, una clase Invoker organizará las llamadas a las distintas operaciones desde nuestro hilo principal.

Strategy:

El patrón strategy permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

Ejemplo de implementación del patrón:

```
4 referencias
abstract class SortStrategy
{
    3 referencias
    public abstract void Sort(List<string> list);
}

1 referencia
class QuickSort : SortStrategy
{
    3 referencias
    public override void Sort(List<string> list)
    {
        Console.WriteLine(" ordenación por quicksort");
    }
}

1 referencia
class ShellSort : SortStrategy
{
    3 referencias
    public override void Sort(List<string> list)
    {
        Console.WriteLine(" ordenación por shellsort");
    }
}
```

```
3 referencias
class SortedListContext
{
    private List<string> list = new List<string>();
    private SortStrategy _sortstrategy;

    1 referencia
    public SortedListContext(List<string> list)
    {
        this.list = list;
    }

    2 referencias
    public void setSortStrategy(SortStrategy sortstrategy)
    {
        this._sortstrategy = sortstrategy;
    }

    2 referencias
    public void Sort()
    {
        this._sortstrategy.Sort(this.list);
    }
}
```

```
0 referencias
static void Main(string[] args)
{
    List<string> listaestudiantes = new List<string>();

    listaestudiantes.Add("Alumno1");
    listaestudiantes.Add("Alumno2");
    listaestudiantes.Add("Alumno3");
    listaestudiantes.Add("Alumno4");

    SortedListContext est = new SortedListContext(listaestudiantes);

    est.setSortStrategy(new QuickSort());
    est.Sort();

    est.setSortStrategy(new ShellSort());
    est.Sort();

}
```

Definimos una clase abstracta y dos clases para la implementación de dos diferentes métodos de ordenación, que heredan de dicha clase abstracta.

Se crea una clase contexto para centralizar la aplicación de una ordenación concreta y finalmente desde el cuerpo principal se organiza la creación del objeto contexto, sobre el cual se aplicarán las diferentes opciones de ordenación.

5.3. Inyección de dependencias

5.3.1. Introducción

La inyección de dependencias (DI, por sus siglas en inglés) es un patrón usado en el diseño orientado a objetos de una aplicación.

DI tiene como finalidad mantener los componentes o capas de una aplicación lo más desacopladas posible. Busca que sea mucho más sencillo reemplazar la implementación de un componente por otro.

Para cumplir con dicho objetivo, DI nos permite inyectar comportamientos a componentes haciendo que nuestras piezas de software sean independientes y se comuniquen únicamente a través de una interface. Esto extrae responsabilidades a un componente para delegarlas en otro, estableciendo un mecanismo a través del cual el nuevo componente puede ser cambiado en tiempo de ejecución.

<u>Ejemplo de implementación de DI simple:</u>

En nuestro ejemplo, vamos a gestionar el comportamiento de un soldado en una batalla. Partimos inicialmente de 3 posibles armas: Revolver, Rifle y Escopeta y nuestro soldado tiene suficiente conocimiento para disparar cada una de ellas:

```
0 referencias
public class Revolver
{
    0 referencias
    public string Disparar()
    {
        return "Pum Pum ..";
    }
}

0 referencias
public class Rifle
{
    0 referencias
    public string Disparar()
    {
        return "Pum pum pum pum pum ..";
    }
}

0 referencias
public class Escopeta
{
    0 referencias
    public string Disparar()
    {
        return "pum PUMMM !! ..";
    }
}
```

```
0 referencias
public class Soldado
{
    0 referencias
    public string DispararRevolver()
    {
        return new Revolver().Disparar();
    }

    0 referencias
    public string DispararRifle()
    {
        return new Rifle().Disparar();
    }

    0 referencias
    public string DispararEscopeta()
    {
        return new Escopeta().Disparar();
    }
}
```

En este momento, nos encontramos con la problemática de que el soldado implementa el mismo la dependencia de cada arma y esto, nos afectará negativamente si por ejemplo,

queremos agregar más armas, ya que tendríamos que modificar la clase Soldado para cada una de ellas.

Solución: Implementar una interface “Arma” y que el constructor de la clase Soldado la reciba como parámetro:

```
3 referencias
public interface IArma
{
    6 referencias
    string Disparar();
}

1 referencia
public class Revolver : IArma
{
    6 referencias
    public string Disparar()
    {
        return "Pum Pum ..";
    }
}

1 referencia
public class Rifle : IArma
{
    6 referencias
    public string Disparar()
    {
        return "Pum pum pum pum pum ..";
    }
}

1 referencia
public class Escopeta : IArma
{
    6 referencias
    public string Disparar()
    {
        return "pum PUMMM !! ..";
    }
}
```

```
1 referencia
public class Soldado
{
    protected IArma arma;

    0 referencias
    public Soldado(IArma _arma)
    {
        this.arma = _arma;
    }

    0 referencias
    public string Disparar()
    {
        return this.arma.Disparar();
    }
}
```

Y en la llamada al objeto soldado, pasaríamos por parámetro el arma que deseamos que dispare:

```
class Program
{
    0 referencias
    static void Main(string[] args)
    {
        Soldado soldado;
        soldado = new Soldado(new Revolver());
        Console.WriteLine(soldado.Disparar());
        soldado = new Soldado(new Escopeta());
        Console.WriteLine(soldado.Disparar());
        soldado = new Soldado(new Rifle());
        Console.WriteLine(soldado.Disparar());
    }
}
```

Para lograr esta tarea DI se basa en un patrón más genérico llamado Inversión de Control (Inversion of control).

Inversion of control (en adelante Ioc) se denomina al proceso en el que un componente invoque nuestro código por nosotros. En .Net, hay varias alternativas para conseguir esto, una de ellas es Autofac.

5.3.2. DI con Ioc Autofac

Autofac es un contenedor IoC para aplicaciones .NET que administra las dependencias entre clases para que las aplicaciones sean fáciles de cambiar a medida que crecen en tamaño y complejidad. Esto se logra tratando las clases regulares de .NET como componentes.

Ejemplo de DI con Autofac:

Para usar Autofac en su proyecto, en primer debemos instalar Autofac desde el Administrador de paquetes NuGet.

Una vez instalado Autofac, creamos una aplicación de consola, se agregan 2 interfaces y 2 clases, con los siguientes nombres:IOutput,ConsoleOutput (implementing IOutput),IDateWriter y TodayWriter (implementing IDateWriter)

```
4 referencias
public interface IOutput
{
    2 referencias
    void Write(string content);
}
```

```
3 referencias
public interface IDateWriter
{
    2 referencias
    void WriteDate();
}
```

```
1 referencia
public class ConsoleOutput : IOutput
{
    2 referencias
    public void Write(string content)
    {
        Console.WriteLine(content);
    }
}
```

```
public class TodayWriter : IDateWriter
{
    private IOutput _output;

    0 referencias
    public TodayWriter(IOutput output)
    {
        _output = output;
    }

    2 referencias
    public void WriteDate()
    {
        _output.Write(DateTime.Today.ToShortDateString());
    }
}
```

Creamos en la clase program el siguiente código:

```
namespace DI_Autofact_Writer
{
    0 referencias
    class Program
    {
        2 referencias
        private static IContainer Container { get; set; }

        0 referencias
        static void Main(string[] args)
        {
            var builder = new ContainerBuilder();
            builder.RegisterType<ConsoleOutput>().As<IOutput>();
            builder.RegisterType<TodayWriter>().As<IDateWriter>();
            Container = builder.Build();

            WriteDate();
        }

        1 referencia
        public static void WriteDate()
        {
            using (var scope = Container.BeginLifetimeScope())
            {
                var writer = scope.Resolve<IDateWriter>();
                writer.WriteDate();
            }
        }
    }
}
```

Cuando se ejecuta, la salida debe ser la fecha actual en la consola.

Explicamos lo que está pasando en el código:

1. Al inicio de la aplicación, estamos creando un ContainerBuilder y registrando nuestros Componentes con él. Un componente en términos simples es un tipo .NET que implementa una interfaz y, por lo tanto, expone servicios.
2. Luego registramos nuestros componentes (clases) con los servicios (interfaces) que exponen. Cuando se registra, Autofac sabe qué instancia de una clase crear cuando debe resolver una interfaz.
3. Finalmente, cuando ejecutamos el programa:
    - El método WriteDate() (en Main()) le pide a Autofac un IDateWriter .
    - Autofac ve que IDateWriter se asigna a TodayWriter por lo que comienza a crear un TodayWriter .
    - Autofac ve que el TodayWriter necesita una IOutput en su constructor.
    - Autofac ve que IOutput se asigna ConsoleOutput por lo que crea una nueva instancia de ConsoleOutput .
    - Autofac usa la nueva instancia de ConsoleOutput para terminar de construir el TodayWriter .
    - Autofac devuelve el TodayWriter totalmente construido, listo para consumir su método WriteDate().

6. Introducción al Desarrollo WEB

6.1. Principales características de un desarrollo WEB

Desarrollo web es un término que define la creación de aplicativos que serán renderizados/visualizados en un navegador web.

Para conseguirlo se hace uso de tecnologías de software del lado del servidor y del cliente que involucran una combinación de procesos que, comunicándose entre sí, obtienen/muestran información o gestionan ítems de negocio.

Características:

- En un único desarrollo web, pueden intervenir varias tecnologías (tanto de lado cliente, como de lado servidor).
- No necesita instalación ni estar en el equipo desde el cual se está consumiendo. Esto permite que sea accesible desde cualquier sitio y desde cualquier dispositivo.
- Se ejecuta sobre un navegador web y, dependiendo de versión y respeto de los estándares del mismo, la web puede cambiar su visualización.
- Al ser un aplicativo que se ejecuta en un navegador web y que puede perder su información al cambiar de página, se usan mecanismos para conseguir persistir o mantener de manera local la información. Estos mecanismos se centran para ello en el uso de cookies, localstorage y sessionstorage en el caso de lado cliente y de objetos session en el caso de lado servidor.

6.2. Lado Cliente/Lado Servidor

En un desarrollo web, se distingue entre lado Cliente y lado Servidor en función de dos aspectos fundamentales:

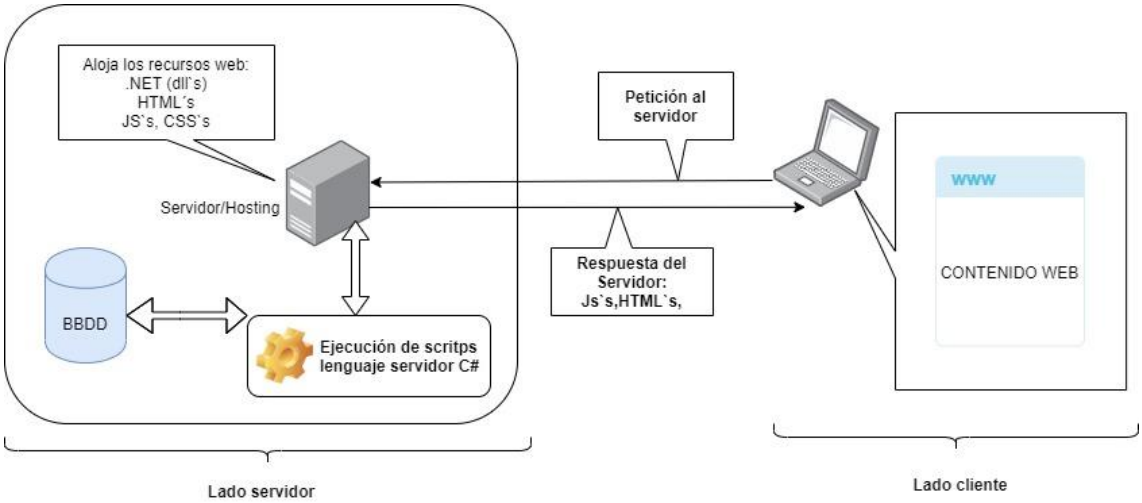
- Sitio donde se ejecuta el código
- Tecnologías usadas

Si nos referimos al sitio donde se ejecuta el código, distinguimos entre el backend, ejecutado en el servidor (lado servidor) y el frontend que se ejecuta en el navegador del usuario (lado cliente).

Si nos referimos a las tecnologías usadas, Los contenidos estáticos como los clásicos elementos de HTML o las imágenes, sencillamente se envían al navegador y allí se visualizan. Los contenidos dinámicos, como una Wiki, un menú desplegable o cualquier tipo de aplicación solo funcionan por medio de scripts, los cuales se han de ejecutar e interpretar con el lenguaje de programación web correspondiente en el lado del servidor. Es por esto que se diferencia fundamentalmente entre los lenguajes de programación del lado servidor (tecnologías de lado servidor) y los lenguajes del lado cliente (tecnologías de lado cliente).

En las tecnologías de lado servidor, las mayormente usadas son: .NET, JAVA y en menor medida PHP.

En las tecnologías de lado cliente, se encuentran: HTML, Javascript, CSS.



7. SQL

7.1. Introducción

SQL es un lenguaje estándar de programación para el interactuar con BBDD`s relacionales. Con SQL podremos acceder a los elementos de la BBDD, manipular los datos, crear y modificar las estructuras de los elementos del SGBD.

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear sentencias y poder así manipular los datos y estructuras de las bases de datos.

7.1.1. Comandos

Según la funcionalidad, se describen los siguientes comandos:De definición de datos (DDL) y De manipulación de datos (DML) y de Lenguaje de control de datos (DCL). En esa formación estudiaremos DDL y DML

Comandos DDL:

Podremos modificar la estructura de una tabla con sentencias para su creación (CREATE), modificación (ALTER) y borrado (DROP) :

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

```
ALTER TABLE Persons
ADD Email varchar(255);
```

```
DROP TABLE Persons;
```

Comandos DML:

Nos permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos: SELECT, INSERT, UPDATE y DELETE

```
INSERT INTO Persons (PersonId,LastName,FirsName) values (1,'Apellido 1','Nombre 1');

SELECT PersonId from Persons;

UPDATE Persons set Lasname='Nombre de prueba';

DELETE From Persons;
```

7.1.2. Claves

Clave primaria:

La clave primaria, o PRIMARY KEY es el identificador unívoco de cada registro. Sólo puede haber una columna con clave primaria por tabla, y los registros deben ser también únicos, es decir no pueden estar repetidos ni ser nulos. La clave primaria es fundamental para crear relaciones entre varias tablas.

Normalmente el formato que presenta es numérico o "id" o "id_nombretabla". Los valores de los registros suelen ser números enteros que identifican a cada uno de los registros que se crean en la tabla.

Como una ayuda para poder crear una clave primaria sin problemas, podemos poner en la columna de clave primaria la restricción de autoincremento (AUTO_INCREMENT).

El autoincremento hace que de forma predeterminada y automáticamente se asigne un número como el dato de este campo. Los números asignados van aumentando en una unidad cada vez que creamos un nuevo registro. El autoincremento sólo puede aplicarse a la columna que tenga la clave primaria.

```
CREATE TABLE prueba1 (
   id_prueba1 INT NOT NULL AUTO_INCREMENT,
   columna1 VARCHAR(255),
   PRIMARY KEY (id_prueba1)
   )
```

Clave externa:

La clave externa crea una relación entre tablas, de forma que la columna a la que se le aplica se relaciona con la columna de la clave primaria de otra tabla. Por lo tanto para crear una clave externa necesitamos tener otra tabla con una clave primaria.

```
CREATE TABLE libros (
   id_libros INT NOT NULL AUTO_INCREMENT,
   titulo VARCHAR(255),
   comentario TEXT,
   ref_autor INT,
   PRIMARY KEY (id_libros),
   FOREIGN KEY (ref_autor) REFERENCES autores (id_autor)
   )
```

7.1.3. Clausulas

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular: FROM, WHERE, GROUP BY, HAVING, ORDER BY

```
SELECT PersonId FROM Persons WHERE FirsName LIKE '%Man' ORDER BY PersonID;
```

```
SELECT Sum(PersonId) FROM Persons GROUP BY City HAVING SUM(PersonID)>45;
```

7.1.4. Operadores

Operadores lógicos:

Los operadores lógicos comprueban la veracidad de alguna condición: AND, OR, NOT

```
SELECT * FROM Apuestas where (idticket>100 AND idOferta=124) OR (idOferta is null) AND idticket is NOT null
```

Operadores de comparación:

Los operadores de comparación comprueban si dos expresiones son iguales: =, >, <, >=, <=, <>, between, like, In

```
SELECT * FROM Apuestas where (idticket>100 AND idOferta=124) OR (idOferta is null) AND idticket is NOT null
```

7.1.5. Funciones de Agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros: AVG,COUNT,SUM,MAX,MIN

```
SELECT Sum(PersonId) FROM Persons GROUP BY City HAVING SUM(PersonID)>45;
```

7.2. Obtención de datos

Para obtener información de un SGBD se usan las consultas de selección. Esta información es devuelta en forma de conjunto de registros.

La sintaxis básica de una consulta de selección es la siguiente: [SELECT Campos FROM Tabla], en donde campos es la lista de campos que se deseen recuperar y tabla es el origen de los mismos.

Podemos usar un alias para referenciar las tablas usadas en la consulta, de la forma [SELECT t1.Campo1, t1.Campo2 FROM Tabla as t1]

7.2.1. Ordenación de registros:

Se puede especificar el orden en que se desean recuperar los registros de las tablas mediante la claúsula [ORDER BY], de la forma [Selec campos from tabla ORDER BY campos ASC/DESC], En donde Lista de campos representa los campos a ordenar. El tipo de orden (ascendente o descendente) se indicará con las claúsulas ASC (se toma como defecto si no se indica nada) ó DESC

```
SELECT playerid,playername FROM Jugadores order by playername DESC
```

7.2.2. Consulta con predicado:

El predicado se incluye entre la claúsula y el primer nombre del campo a recuperar, para indicar una acción sobre los registros obtenidos. Los posibles predicados son: TOP,DISTINCT

```
SELECT TOP 5 playerid,playername FROM Jugadores
SELECT DISTINCT playerid,playername FROM Jugadores
```

7.2.3. Subconsultas

Una subconsulta es una instrucción SELECT anidada dentro de una instrucción SELECT,SELECT…INTO, INSERT…INTO, DELETE, o UPDATE o dentro de otra subconsulta.

Puede utilizar tres formas de sintaxis para crear una subconsulta:

- comparación [ANY | ALL | SOME] (instrucción sql)
- expresión [NOT] IN (instrucción sql)
- [NOT] EXISTS (instrucción sql)

Se puede utilizar una subconsulta en lugar de una expresión en la lista de campos de una instrucción SELECT o en una cláusula WHERE o HAVING

```
SELECT * FROM  Productos WHERE PrecioUnidad > ANY (SELECT PrecioUnidad FROM DetallePedido WHERE Descuento = 0 .25)
SELECT * FROM  Productos WHERE PrecioUnidad > ALL (SELECT PrecioUnidad FROM DetallePedido WHERE Descuento = 0 .25)
SELECT * FROM  Productos WHERE PrecioUnidad [NOT] IN (SELECT PrecioUnidad FROM DetallePedido WHERE Descuento = 0 .25)
SELECT * FROM  Productos WHERE PrecioUnidad [NOT] EXISTS (SELECT PrecioUnidad FROM DetallePedido WHERE Descuento = 0 .25)

SELECT *,(SELECT PrecioUnidad FROM DetallePedido WHERE Descuento = 0 .25) as campo1 FROM  Productos
WHERE PrecioUnidad EXISTS (SELECT PrecioUnidad FROM DetallePedido WHERE Descuento = 0 .25)
```

7.2.4. Consultas de Unión Internas

Las vinculaciones entre tablas se realizan mediante la cláusula INNER que combina registros de dos tablas siempre que haya concordancia de valores en un campo común. Su sintaxis es: [SELECT campos FROM tb1 INNER [JOIN,LEFT,RIGTH] tb2 ON tb1.campo1 comp tb2.campo2].

La sentencia INNER JOIN es la sentencia JOIN por defecto, y consiste en combinar cada fila de una tabla con cada fila de la tabla referenciada, seleccionado únicamente aquellas filas coincidentes según igualdad en sus claves.

```
SELECT E.Nombre , D.Nombre  FROM Empleados E JOIN Departamentos D ON E.DepartamentoId = D.Id
```

La sentencia LEFT JOIN combina los valores de la primera tabla con los valores de la tabla referenciada. Devolviendo siempre las filas de la primera tabla, aunque no cumplan la condición de igualdad en sus claves.

```
SELECT E.Nombre,D.Nombre FROM Empleados E LEFT JOIN Departamentos D ON E.DepartamentoId = D.Id
```

La sentencia RIGHT JOIN combina los valores de la primera tabla con los valores de la tabla referenciada. Siempre devolverá las filas de la tabla referenciada, incluso aunque no cumplan la condición de igualdad en sus claves.

```
SELECT E.Nombre,D.Nombre FROM Empleados E RIGHT JOIN Departamentos D ON E.DepartamentoId = D.Id
```

Existe otra sentencia, además de las indicadas hasta ahora, que se encarga de mostrar todas las filas de ambas tablas, sin importar que no existan coincidencias (usará NULL como un valor por defecto para dichos casos)

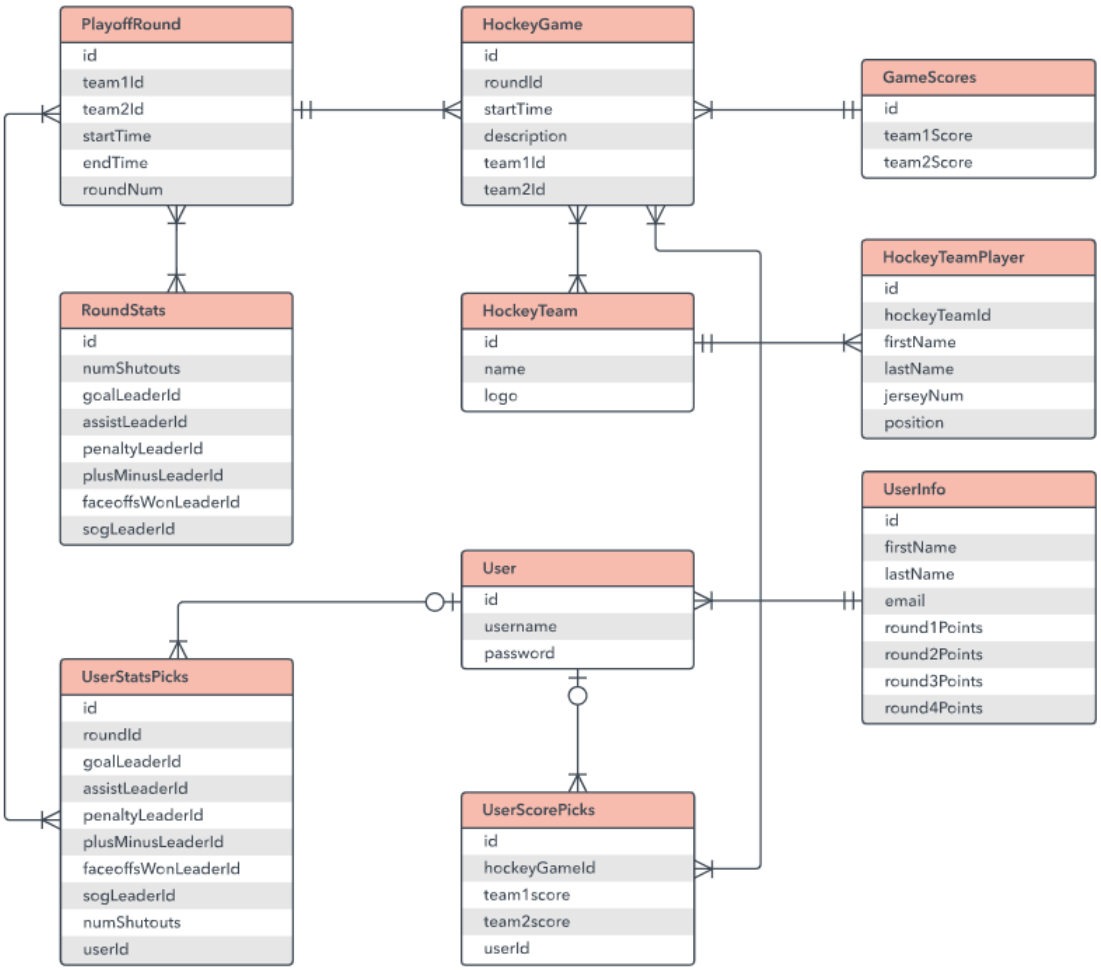
```
SELECT E.Nombre,D.Nombre FROM Empleados E FULL JOIN Departamentos D ON E.DepartamentoId = D.Id
```

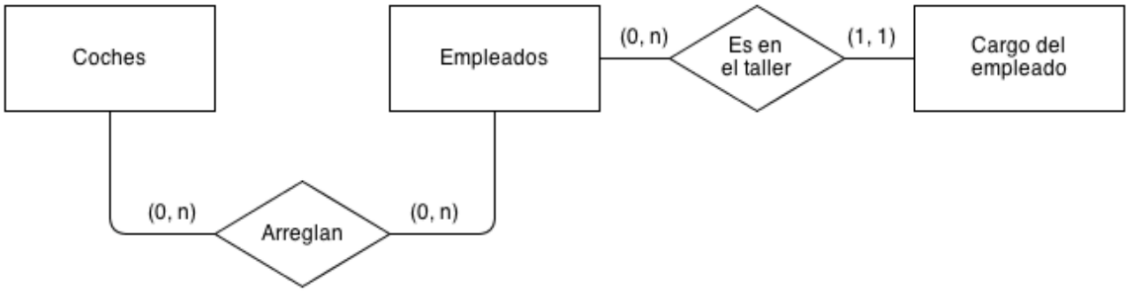
7.3. Diagrama ER

7.3.1. Introducción

¿Qué es un modelo entidad relación?

Un diagrama entidad-relación, también conocido como modelo entidad relación o ERD, es un tipo de diagrama de flujo que ilustra cómo las "entidades", como personas, objetos o conceptos, se relacionan entre sí dentro de un sistema. Los diagramas ER se usan a menudo para diseñar o depurar bases de datos relacionales en los campos de ingeniería de software, sistemas de información empresarial, educación e investigación. También conocidos como los ERD o modelos ER, emplean un conjunto definido de símbolos, tales como rectángulos, diamantes, óvalos y líneas de conexión para representar la interconexión de entidades, relaciones y sus atributos. Son un reflejo de la estructura gramatical y emplean entidades como sustantivos y relaciones como verbos.





7.3.2. Componentes del diagrama ER

Entidad

Las entidades representan cosas u objetos (ya sean reales o abstractos), que se diferencian claramente entre sí.

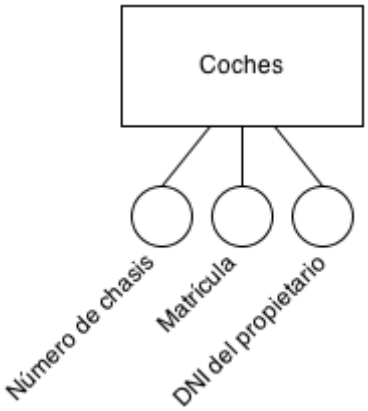
Estas entidades se representan en un diagrama con un rectángulo, como los siguientes.

Atributos

Los atributos definen o identifican las características de entidad (es el contenido de esta entidad). Cada entidad contiene distintos atributos, que dan información sobre esta entidad. Estos atributos pueden ser de distintos tipos (numéricos, texto, fecha...).

Los atributos se representan como círculos que descienden de una entidad, y no es necesario representarlos todos, sino los más significativos, como a continuación.

Los atributos referentes a las claves primarias serán subrayados.

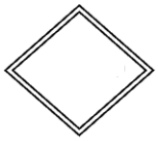


Relación

Es un vínculo que nos permite definir una dependencia entre varias entidades, es decir, nos permite exigir que varias entidades compartan ciertos atributos de forma indispensable.

Las relaciones se muestran en los diagramas como rombos, que se unen a las entidades mediante líneas.

Existen relaciones fuertes y relaciones débiles. Las relaciones débiles son las conexiones entre una entidad débil y su propietario. Se representan con un doble diamante

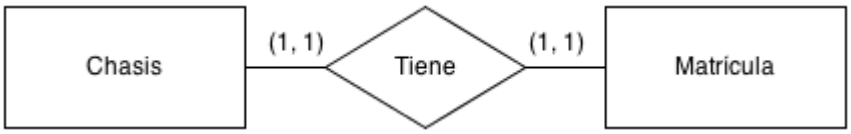

Relación débil

Cardinalidad de las relaciones

Podemos encontrar distintos tipos de relaciones según como participen en ellas las entidades.

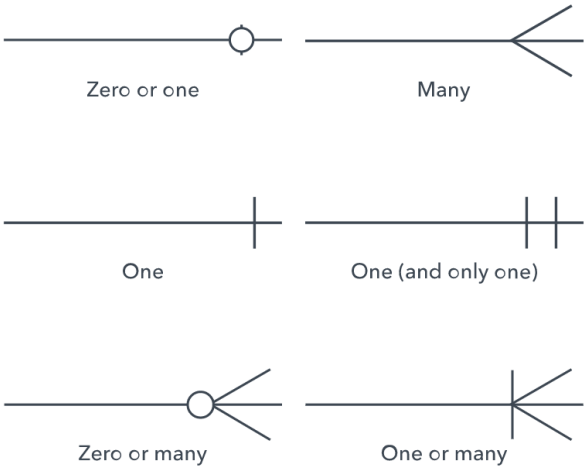
Esto complementa a las representaciones de las relaciones, mediante un intervalo en cada extremo de la relación que especifica cuantos *objetos* o *cosas* (de cada entidad) pueden intervenir en esa relación.

Uno a uno: Una entidad se relaciona únicamente con otra y viceversa.



Uno a varios o **varios a uno**: determina que un registro de una entidad puede estar relacionado con varios de otra entidad, pero en esta entidad existir solo una vez.

Varios a varios: determina que una entidad puede relacionarse con otra con ninguno o varios registros y viceversa.

El primer número de los indicadores numéricos mínimo de registros en una relación y el segundo el máximo (si no hay límite se representa con una "*n*").

Las relaciones de cardinalidad también pueden ser representadas de la siguiente manera: