Un SW es **complejo** porque:

* Hay muchos requerimientos y pueden cambiar durante el proceso.
* Falta de comunicación entre programadores y usuarios.
* Dificultad para gestionar el desarrollo. El SW crece mucho y demanda trabajo extra.
* Permite flexibilidad.
* Problemas de comportamiento ya que pequeñas variaciones en la entrada pueden producir grandes cambios en las salidas y se necesita testear.

Atributos:

* Estructura jerárquica. Cada sistema está formado por subsistemas.
* Se puede modularizar por más que tengan dependencia entre ellos.
* Permite reutilizar código de diversos sistemas.
* Son evoluciones de sistemas simples. Es difícil comenzar de cero uno complejo.

Hay dos jerarquías:

* "parte de" hace referencia a las propiedades de los objetos. Por ejemplo: altura (propiedad) es *parte de* persona (objeto).
* "es un" hace referencia a qué clase de objeto es. Por ejemplo: persona (objeto) *es un* humano (clase).

Las personas no pueden lidiar con sistemas complejos, por lo que se puede descomponer, abstraer o jerarquizar al SW.

Descomponer significa dividirlo en subsistemas lo suficientemente pequeños para entenderlos. Puede ser logarítmica (un módulo – un paso de un proceso) u orientada a objetos (objetos autómatas que colaboran entre sí).

**POO**

Los programas se organizan como colecciones colaborativas de objetos, en la que cada objeto representa la instancia de una clase, y todas las clases pertenecen a una jerarquía de clases unidas por relaciones de herencia.

Elementos principales:

* *Abstracción*: Representa las características esenciales del objeto que lo distingue de los demás, su vista externa, la interfaz. Permite separar el comportamiento esencial de su implementación.
* *Encapsulamiento*: Ocultar la implementación de los comportamientos esenciales del objeto.
* *Modularidad*: Dividir al programa en unidades independientes pero que mantienen conexiones entre sí.
* *Jerarquías*: Permiten ordenar las abstracciones como si fueran una sola.

**OBJETO**

Es una entidad o instancia, real o abstracta, compuesta por:

* Estado: Representa todas las propiedades con sus respectivos valores.
* Comportamiento: Son métodos que determinan cómo actúa y reacciona el objeto según su estado.

Métodos:

* Modificadores de acceso:
* Public: Sin restricciones.
* Private: Sólo desde la clase.
* Internal: Desde el módulo actual.
* Protected: Desde la clase y sus subclases.
* Protected internal: Desde la clase, sus subclases y el módulo actual.

Para las clases sólo public, internal o static. Para los miembros, todas.

* Selectores e iteradores: Acceden al estado sin alterarlo.
* Constructores: Inicializa o crea el objeto. Se llama igual que la clase en la que está. No debe retornar nada. Puede haber varios constructores con distintos parámetros, es decir que permite overloading.

*public Posicion(int x, int y)*

*{*

*this.x = x;*

*this.y = y;*

*}*

* Destructores: Libera el estado del objeto y lo destruye. Se ejecuta cuando el objeto es recolectado por el *garbage collector*. Tiene el mismo nombre que la clase, pero precedido por ~. No toma parámetros ni retorna algo.

*~Posicion() { //Liberar las cosas }*

* Identidad: Es la propiedad que lo distingue de los demás objetos.

Los objetos interactúan mediante el envío de mensajes entre ellos.

Mensaje: Solicitud del emisor para que se ejecute una acción. El receptor ejecuta métodos como respuesta. El envío de un mensaje soporta la abstracción, es decir que se puede cambiar la implementación de un objeto sin que varíe su interfaz.

Para crear una instancia u objeto se define una variable de ese tipo y se utiliza la palabra *new*:  
 *Posicion posicion1 = new Posicion();*

**CLASES**

Conjunto de objetos que comparten una estructura, comportamiento y semántica. Representa la abstracción de las características comunes a todos los objetos de esa clase.

Se puede dividir en:

* Interfaz (vista externa): Es la declaración de todas las operaciones y propiedades de la clase.
* Implementación (vista interna): Son los detalles para implementar las operaciones de la interfaz.

Características:

* Campos: Variables con los datos del objeto.

*private string \_nombre;*

* Propiedades: Proveen acceso de L/E a los elementos del objeto. Permiten encapsular el estado.

*public string nombre  
{*

*get { return \_nombre; }* Lectura *set { \_nombre = value; }* Escritura

*}*

* Métodos: Acciones que puede hacer el objeto. Puede pedir parámetros o no, como también puede retornar algo o no(void).

*public int MostrarNumero(int numero)*

*{*

*return numero;*

*}*

* Eventos: Acciones que se pueden responder o manejar.
* Overloading (Sobrecarga): Varios métodos en una misma clase con el mismo nombre, diferenciados por la cantidad y tipo de parámetros que recibe.

**Abstract class**

* No se puede instanciar. Se van a instanciar sus objetos desde las clases derivadas.
* Se usa especialmente para heredar. Se pueden crear en ella propiedades y métodos sin implementación ya que eso se hará en la clase derivada mediante la palabra *override*.

*public string nombre { get; }* Para que sea de solo lectura

*public abstract bool Hacer(int x);*

**Static class**

* No se puede utilizar para crear instancias de este tipo de clase.
* Se utiliza sólo en el módulo declarado.
* No se utiliza para heredar.
* Compatible solo con objetos estáticos.

El Patrón Singleton asegura que para una clase haya solo una instancia, por lo que utiliza objetos de tipo static. Consiste en tener un constructor de clase normal, y una propiedad y un método estáticos. En el método se controlará que se instancie una sola vez los parámetros del constructor.

**Relaciones entre clases**

* Asociación: Es una dependencia entre dos clases, es decir que una clase está ubicada en otra clase, y viceversa (Casilla está ubicada en una Posición y una Posición está ubicada en una Casilla). En ambas clase tiene que haber un campo que relacione a la otra clase. Tener en cuenta la cardinalidad.
* Agregación: Todo/Parte de. Una clase es parte de otra clase (los Alumnos son parte de una Clase). Es una dependencia de uno a muchos. No se hacen responsables del ciclo de vida.
* Composición: Igual que la agregación, solo que el todo es responsable del ciclo de vida de las partes, es decir que puede construirlas y destruirlas.
* Herencia: Es una relación de generalización/especialización. La subclase tendrá todas las características de la superclase más sus características propias. Puede ser simple o múltiple (en C# lo más cercano a herencia múltiple es Interfaz).

**OVERRIDING**

Permite redefinir los métodos de una clase en su subclase. Sujeto a polimorfismo.

En la superclase se declara al método como:

*public* ***virtual*** *string Mostrar(string nombre) { …}*

Y en la subclase se hace la sobreescritura llamando al método como:

*public* ***override*** *string Mostrar(string nombre) { ,,, }*

**POLIMORFISMO**

A partir de la herencia, permite crear una instancia de una clase derivada y asignarla a una variable de la clase base.

Si se llama a un método sobreescrito, asegura que se ejecute la versión correcta del mismo.

**Sobrecarga de operadores**

Permite definir el comportamiento de los operadores del lenguaje cuando interactúan con las clases.

**DELEGADOS**

Objetos que encapsulan referencias a métodos. Al definirlo hay que especificar la firma y el tipo de retorno.

Definición: *public delegate int NombreDelegado(int parámetro);*

Si se usa como parámetro un derivado de EventArgs:   
 *public delegate void NombreDelegado(object sender, DerivadoEventArgs e);*

Cuando se quiera invocar al método, se lo hace por medio del delegado.

Los delegados permiten pasar los métodos como parámetros a los métodos. Se puede llamar a varios métodos en un solo evento. No es necesario que los métodos coincidan exactamente con el tipo de delegado.

**EVENTOS**

Se disparan cuando un objeto quiere indicarle a otro que invoque algún método suyo cuando ocurre un determinado suceso. Para indicar qué evento debe invocar el emisor del evento, se define un delegado.

Definición: *public event TipoDelegado NombreEvento;*

**EventHandler**

Es un delegado que representa el método que controlará eventos que no tienen datos de evento.   
Definición: *public delegate void EventHandler( object sender, EventArgs e )*

*Implementación:*

*public event EventHandler NombreEvento;  
 public void Método()  
 {  
 if (NombreEvento != null)  
 NombreEvento(this, new EventArgs e);  
 }*

**EventArgs**

Representa la clase base para las clases que contienen datos de eventos y proporciona un valor para utilizar en los eventos que no incluyen datos de evento.

*public class PersonaEventArgs : EventArgs  
{  
 public string nombre { get; set; }  
 public int edad { get; set; }  
 public PersonaEventArgs(string nombre, int edad)  
 {  
 this.nombre = nombre;   
 this.edad = edad;*

*}*

*}*

public class Persona  
 {  
 public event EventHandler<PersonaEventArgs> DatosPersona;  
 *public void Método()  
 {  
 if (NombreEvento != null)  
 NombreEvento(this, new PersonaEventArgs (nombre, edad));  
 }  
 }*

**INTERFAZ**

Es un conjunto de propiedades y métodos que serán implementados por otra clase. Define el comportamiento de una clase cuando ésta la implementa. Representa un contrato con la clase.

Desacoplamiento:

Al instanciar con una interfaz como tipo de la variable, se permite trabajar con cualquier clase que la implemente.

Gestión de dependencia:

Si se utiliza la interfaz como parámetro en un método, éste aceptará objetos de cualquiera de las clases que la implemente.

Herencia múltiple de interfaces:

Una clase puede heredar de otra clase y, además, de una o varias interfaces. La clase deberá implementar todos los miembros definidos en la interfaz.

**ARQUITECTURA DE SW**

Tiene que ver con el diseño y la implementación de estructuras de SW de alto nivel de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, la confiabilidad, escalabilidad, portabilidad, y disponibilidad.

**Monolítica**

* El software se estructura en unidades funcionales muy acopladas. No hay separación física ni lógica.
* El sistema final consiste de módulos compilados separadamente y luego enlazados.
* Es eficiente ya que se producen pocos cambios de contexto pero tienen baja reutilización, poca escalabilidad y los cambios afectan a todo el sistema.

**Cliente-Servidor**

* Modelo de aplicación distribuida donde se reparten las tareas entre los proveedores de servicios (servidores) y quienes los usan (clientes).
* La separación entre cliente y servidor puede ser solo a nivel lógico, donde ambos programas corren en el mismo equipo; o físico, donde cada uno de los programas corre en equipos diferentes. Esta última es la disposición más común.
* Se centraliza el control a accesos, recursos e integridad de datos. Permite escalabilidad independiente en cliente y servidor y un mantenimiento más sencillo. La desventaja es que haya muchos clientes y el servidor no lo soporte.

**MultiCapas**

* El objetivo principal es separar la lógica de negocio de la lógica de diseño.
* Cada capa tiene asignada una funcionalidad simple e interactúa con las demás capas a través de una API. Si hay un problema con alguna capa en particular se corrige teniendo en cuenta solo de mantener la API.
* El diseño más utilizado actualmente es el de 3 capas:
* Capa de presentación: Es la que ve el usuario. Se comunica únicamente con la de negocios.
* Capa de negocios: Se reciben peticiones del usuario y se da respuesta. Aquí se establecen todas las reglas que deben cumplirse a nivel de negocio.
* Capa de datos: Encargada de recibir peticiones para almacenar y recuperar datos.
* Los términos capas y niveles no significan lo mismo, una capa hace referencia a como es segmentada una solución a nivel lógico, mientras que un nivel hace referencia a la segmentación física.

**Orientada a servicios (SOA)**

* Utiliza servicios para dar soporte a los requisitos del negocio.
* Brinda una forma bien definida de exposición e invocación de servicios para facilitar la interacción entre procesos propios y de terceros.
* Los servicios son débilmente acoplados y altamente escalables.

**PARADIGMAS DE PROGRAMACIÓN**

**Programación imperativa:** Los programas estás constituidos por sentencias que alteran el estado del mismo. Las asignaciones producen una serie de efectos laterales que oscurecen la semántica del lenguaje:

* Es difícil demostrar la corrección de un programa
* No es posible evaluar expresiones en paralelo
* Hay complicaciones para optimizar código

**Programación funcional:** El modelo funcional tiene como objetivo la utilización de funciones matemáticas puras sin efectos laterales y, por tanto, sin asignaciones destructivas.

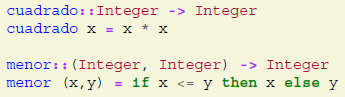
Características:

* Transparencia referencial: el valor que devuelve una función está únicamente determinado por el valor de sus argumentos.
* Funciones de orden superior: permite que las funciones sean tratadas como valores de primera clase.
* Inferencia de tipos y polimorfismo.
* Lazy evaluation: una expresión no se evalúa hasta que es necesario

**HASKEL**

**Funciones**

* Las funciones son valores, pueden ser transferidas a otras funciones.
* Las funciones que reciben otras funciones como argumentos se denominan funciones de orden superior.
* Se admiten funciones anónimas.
* Declarar funciones:





* El operador -> es asociativo a derecha. El último parámetro es el tipo de retorno de la función.



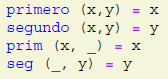
* La aplicación de funciones es asociativa a izquierda.
* Los operadores infijos son en realidad funciones, entonces se pueden aplicar parcialmente.



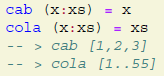
* **Funciones recursivas**



**Pattern matching:** Permite clarificar la definición de funciones cotejando sus argumentos con patrones.



Se pueden aplicar a listas:



**Guardias:** Facilita el entendimiento de funciones que usan múltiples condiciones.



**Listas por comprensión:** Permite la generación de listas a partir de otras existentes.



Pueden usarse múltiples generadores separados por “,”:



Generadores dependientes de alguno anterior:



Filtro en List comprehensions:

