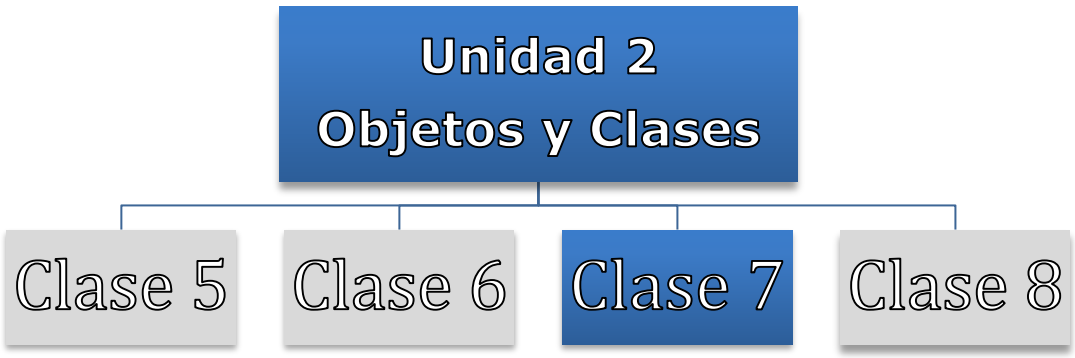
PROGRAMACIÓN ORIENTADA A OBJETOS



Docente titular y autor de contenidos: Prof. Ing. Darío Cardacci

Presentación

En la clase seis haremos énfasis en las relaciones de agregación y asociación. De esta manera podremos potenciar nuestros desarrollos.

También analizaremos la forma de implementar eficientemente en código estas relaciones.

Esperamos que luego de analizar estos temas Ud. pueda percibir el aporte que los mismos les otorgan a los desarrollos orientados a objetos.

Por todo lo expresado hasta aquí es que esperamos que usted, a través del estudio de esta unidad logre:

Los siguientes **contenidos** conforman el marco teórico y práctico de esta clase. A partir de ellos lograremos alcanzar el resultado de aprendizaje propuesto. En negrita encontrará lo que trabajaremos en la clase 7.

- Eventos. Suscripción a eventos. Suscripción a eventos utilizando el IDE. Suscripción a eventos mediante programación. Suscripción a eventos mediante métodos anónimos. Publicación de eventos. Desencadenar eventos.
- Modificadores de acceso. Clases abstractas, selladas y estáticas. Miembros estáticos en clases estáticas.
- Relaciones básicas entre clases. "Generalización-Especialización", "Parte de".
- Relaciones derivadas entre clases. Herencia. Herencia simple. Herencia múltiple. Teoría de Tipos. Tipos anónimos.
- Sobrescritura de métodos. Métodos virtuales. Polimorfismo.
- **Agregación. Simple y con contención física.**

- **Asociación y relación de Uso.**
- **Elementos que determinan la calidad de una clase: acoplamiento, cohesión, suficiencia, compleción y primitivas.**
- **Relaciones entre objetos: enlace y agregación.**
- **Acceso a la clase base desde la clase derivada.**
- **Acceso a la instancia actual de la clase.**

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta clase. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor tutor.

Índice de contenidos y Actividades

- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/access-keywords

Para el estudio de estos contenidos usted deberá consultar la bibliografía que aquí se menciona:

BIBLIOGRAFÍA RECOMENDADA

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007.

Links a temas de interés

Desarrollo .NET

https://docs.microsoft.com/en-us/dotnet/csharp/index

https://docs.microsoft.com/en-us/dotnet/standard/get-started

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

1. Agregación

La **agregación** es una relación entre clases que posee su sustento en la Jerarquía Todo-Parte que plantea el modelo orientado a objetos. Este tipo de relación permite que una clase contenedora que representa al todo, agregue una o más clases que representan las partes. Esto hace posible concretar estructuras complejas agrupando estructuras sencillas.

Este tipo de relación se define a nivel de clases pero se manifiesta a nivel de objetos. Al momento que generar una relación de agregación se puede dar de dos tipos. A la primera la denominamos simplemente **agregación** y a la segunda **composición**.

La diferencia entre ambas es que en la **agregación** la clase contenedora o que agrega no determina el ciclo de vida de lo agregado, mientras que en la **composición** sí.

Cuando decimos que la clase contenedora determina el ciclo de vida de lo contenido, establecemos que la clase contenedora crea las instancias que se necesitan y también al finalizar su existencia, finaliza la existencia de lo agregado.

En el ejemplo **Ej0023** se observan las clases **Accionista, Accion, AccionEmpresaA y AccionEmpresaB.**

```
public class Accionista
{
    1 referencia
    public Accionista(string pNombre) { this.Nombre = pNombre; }
    1 referencia
    public string Nombre { get; set; }
    private List<Accion> ListaDeAcciones = new List<Accion>();
    1 referencia
    public void AgregarAccion(Accion pAccion) {this.ListaDeAcciones.Add(pAccion); }
    1 referencia
    public void BorrarAccion(Accion pAccion) {this.ListaDeAcciones.Remove(pAccion);}
    6 referencias
    public List<Accion> VerAcciones() {return this.ListaDeAcciones; }
}
```

EJ0023

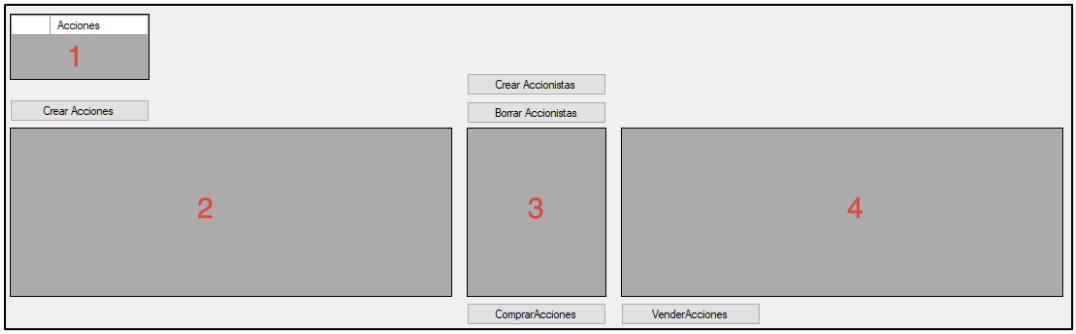
En este ejemplo un **Accionista** agrega muchas acciones. Esto lo puede realizar debido a que la clase accionista implemente una lista de acciones.

```
abstract public class Accion
    2 referencias
{   public Accion (int pCantidad,string pIdentificador,string pDescripcion,decimal pCotizacion)
    {   this.Cantidad = pCantidad;this.Identificador = pIdentificador;
        this.Descripcion = pDescripcion;this.Cotizacion = pCotizacion;
    }
    1 referencia
    public int Cantidad { get; set; }
    1 referencia
    public string Identificador { get; set; }
    1 referencia
    public string Descripcion { get; set; }
    1 referencia
    public decimal Cotizacion { get; set; }
}
2 referencias
public class AccionEmpresaA : Accion
    1 referencia
{ public AccionEmpresaA(int pCantidad, string pIdentificador, string pDescripcion, decimal pCotizacion)
        : base(pCantidad, pIdentificador, pDescripcion, pCotizacion) { }
}
2 referencias
public class AccionEmpresaB : Accion
    1 referencia
{ public AccionEmpresaB(int pCantidad, string pIdentificador, string pDescripcion, decimal pCotizacion)
      : base(pCantidad, pIdentificador, pDescripcion, pCotizacion) { }
}
```

Ej0023

La instanciación de los accionistas y de las acciones se producen en momentos distintos. Esto corrobora que los ciclos de vida de ambos son distintos.

Un accionista puede ser creado o borrado, aún cuando este deja de existir, las acciones que tenía siguen existiendo. Claramente en el ejemplo **Ej0023** se aplica el concepto tradicional de **agregación.**



Ej0023

La grilla 1 posee los dos tipos de acciones que se pueden crear, la grilla 2 muestra las acciones creadas, la grilla 3 expone los accionistas creados y la grilla 4 muestra las acciones del accionista seleccionado en la grilla 3.

A continuación se exponen los fragmentos de códigos utilizados para el ejemplo **Ej0023.** El primero muestra como se configuran las grillas.

```
public Form1()...
List<Accionista> ListaAccionista = new List<Accionista>();
List<Accion> ListaAccion = new List<Accion>();
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    #region "Configuración de la Grilla"

    this.dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView2.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView3.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView4.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView1.Rows.Add(2);
    this.dataGridView1.Columns[0].Width = 100;
    this.dataGridView1.Rows[0].Cells[0].Value="Acción Empresa A";
    this.dataGridView1.Rows[1].Cells[0].Value = "Acción Empresa B";
    #endregion
}
1 referencia
```

Ej0023

El código que se expone a continuación es el que permite crear las acciones.


```
public Form1()...
List<Accionista> ListaAccionista = new List<Accionista>();
List<Accion> ListaAccion = new List<Accion>();
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    #region "Configuración de la Grilla"

    this.dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView2.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView3.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView4.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView1.Rows.Add(2);
    this.dataGridView1.Columns[0].Width = 100;
    this.dataGridView1.Rows[0].Cells[0].Value="Acción Empresa A";
    this.dataGridView1.Rows[1].Cells[0].Value = "Acción Empresa B";
    #endregion
}
1 referencia
```


Ej0023

A continuación el código que permite crear **Accionistas.**

```
private void button4_Click(object sender, EventArgs e)
{
    this.ListaAccionista.Add(new Accionista(Interaction.InputBox("Nombre: ")));
    this.dataGridView3.DataSource = null; this.dataGridView3.DataSource = this.ListaAccionista;
}
```

Ej0023

El siguiente fragmento de código permite que un accionista pueda adquirir acciones.

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        // Se agrega la acción comprada al accionista que la compro
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).AgregarAccion(
       (Accion)(this.dataGridView2.SelectedRows[0].DataBoundItem));
        // Se retira la accion de la lista de acciones disponibles
        this.ListaAccion.Remove((Accion)(this.dataGridView2.SelectedRows[0].DataBoundItem));
        // Se refresca la lista que muestra las acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se muestra las acciones del Accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    { }
}
```

Ej0023

A continuación el código que permite que un accionista venda sus acciones.

```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        // Se selecciona la acción de la grilla de acciones que ya fueron compradas
        Accion A = (Accion)(this.dataGridView4.SelectedRows[0].DataBoundItem);
        // Se saca la accion de la lista de acciones del accionista
        ((Accionista)(this.dataGridView3.SelectedRows[0].DataBoundItem)).BorrarAccion(A);
        // Se agrega la accion a la lista de acciones disponibles
        this.ListaAccion.Add(A);
        // Se refresca la grilla de acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se refresca la grilla de acciones del accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
       ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();

    }
    catch (Exception)
    { }
}
```

Ej0023

El siguiente código es el que permite borrar a un accionista.

```
private void button5_Click(object sender, EventArgs e)
{
    try
    {
        // si posee acciones se pasan como disponibles
        Accionista ACC = (Accionista)(this.dataGridView3.SelectedRows[0].DataBoundItem);
        if(ACC.VerAcciones().Count>0)
        {
            this.ListaAccion.AddRange(ACC.VerAcciones());
        }
        // Se borra el accionista
        this.ListaAccionista.Remove(ACC);
        // Se refresca la grilla de acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se refresca la grilla de accionistas
        this.dataGridView3.DataSource = null; this.dataGridView3.DataSource = this.ListaAccionista;
        // Se refresca la grilla de acciones del accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
            ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    {}
}
```

Ej0023

Finalmente el código que permite que cuando se selecciona un accionista, se actualice la grilla que contiene las acciones que están en su posesión.

```
private void dataGridView3_CellEnter(object sender, DataGridViewCellEventArgs e)
{
    try
    {
        // Se muestra las acciones del Accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    {}
}
```

Ej0023

La **composición,** como se expresó anteriormente, se manifiesta en relaciones donde los ciclos de vida del que agrega y los agregados estan relacionados. Esto significa que cuando se crea al que agrega, se crean los agregados. Cuando programamos, para poder hacer uso de esta posibilidad se debe acudir al uso de los constructores y los finalizadores.

A modo de ejemplo, supongamos que poseemos una máquina que está compuesta por tres elementos constitutivos y propios. El primer elemento sirve para introducirle el input o cantidad de materiales que le suministramos, se

denominará *Entrada*. El segundo, es el que realiza el proceso o sea que produce los productos terminados que comercializamos, llevará ese mismo nombre *Proceso*. Finalmente la tercera, nos informa cuanto producto terminado obtuvimos en base al input suministrado y cuanto material sobró si este fuera el caso, se denominará *Salida*.

En particular consideremos que nuestra máquina produce bronce y para lograrlo se mezcla 90% de cobre y 10% de estaño. En el ejemplo **Ej0024** vemos implementado el ejemplo.

```
3 referencias
public partial class Form1 ...
4 referencias
public class Maquina...
4 referencias
public class Entrada...
3 referencias
public class Proceso...
2 referencias
public class Salida...
```

Ej0024

El ejemplo utiliza cinco clases, **Máquina** quien compone (agrega condicionando los ciclos de vida de los agregados). **Entrada**, **Proceso**, **Salida** que representan las clases agregadas. Finalmente la clase **Form1** que es la que instancia a la máquina, la utiliza y define el fin de su existencia.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {InitializeComponent();}
    Maquina M =null;
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {M = new Maquina(); }
    1 referencia
    private void button2_Click(object sender, EventArgs e)
    { M.Dispose();}
    1 referencia
    private void button3_Click(object sender, EventArgs e)
    {M.Procesar();}
}
```

Ej0024

La clase **Maquina** posee la siguiente implementación:

```
public Maquina()
{
    //Constructor donde se instancian los agregados
    E = new Entrada(decimal.Parse(Interaction.InputBox("Cantidad ingresada de cobre en Kg: ")),
                    decimal.Parse(Interaction.InputBox("Cantidad ingresada de estaño en Kg: ")));
    P = new Proceso();
    S = new Salida();
    MessageBox.Show("Se han creado los objetos Entrada, Proceso y Salida !!!");
}
1 referencia
public void Procesar()
{
    P.ProducirBronce(E);
    S.VerResultados(P);
}
1 referencia
public void Dispose()
{
    //Finalizador que puede ser llamado por el usuario. El mismo rompe la composición
    this.E = null; this.P = null; this.S = null;
    MessageBox.Show("Dispose: Se han destruido los objetos Entrada, Proceso y Salida !!!");
    this.SeUsoDispose = true;
}
0 referencias
~Maquina()
{
    //Finalizador que es llamdo por el Garbage Colector. El mismo rompe la composición en caso que no se
    //haya roto por el llamado al Dispose
    if (!this.SeUsoDispose)
    {
        this.E = null; this.P = null; this.S = null;
        MessageBox.Show("Garbage Collector: Se han destruido los objetos Entrada, Proceso y Salida !!!");
    }
}
```

Ej0024

En el contructor de la clase **Maquina**, se observa la instanciación de los tres objetos que se necesitan para trabajar. Un objeto **Entrada,** un objeto **Proceso** y un objeto **Salida.** Luego el método **Procesar** de **Maquina** es utilizado el objeto **Proceso,** al cual se le solicita **ProducirBronce** y se le envía como parámetro el objeto **Entrada.** Finalmente el mismo método (**Procesar**), utiliza el objeto **Salida** y le solicita **VerResultados**, enviándole como parámetro el objeto **Proceso.**

También se puede observar en la clase **Maquina** el método **Dispose** y el finalizador **~Maquina**. En ellos es donde se predispone la situación para que se produzca la finalizción del ciclo de vida de los tres objetos: **Entrada, Proceso** y **Salida**, haciendo que las variables que lo apuntaban: **E, P** y **S** respectivamente, apunten a **null.** Esto produce que los objetos queden desapuntados, por lo tanto se transforman en basura dentro de la memoria, para que el garbage collector los elimine, generando la finalización de sus ciclos de vida.

Queda explísita la creación de los objetos agregados en el constructor de la clase **Maquina,** quien genera la composición, y en ella misma la finalización de los ciclos de vida de los agregados, **Entrada, Proceso y Salida,** en los finalizadores de la clase que les dio origen.

A continuación los fragmentos de código de las clases **Entrada**, **Proceso** y **Salida.**

```
public class Entrada
{
    private decimal VCantidadCobreKg = 0;
    private decimal VCantidadEstanoKg = 0;
    1 referencia
    public Entrada(decimal pCantidadCobreKg, decimal pCantidadEstanoKg)
            { this.CantidadCobreKg = pCantidadCobreKg; this.CantidadEstanoKg = pCantidadEstanoKg; }
    3 referencias
    public decimal CantidadCobreKg
    {
        get { return VCantidadCobreKg; }
        set {
            if (value < 1) { MessageBox.Show("La cantidad debe ser mayor a 1 Kg"); }
            else { this.VCantidadCobreKg = value; }
            }
    }
    3 referencias
    public decimal CantidadEstanoKg
    {
        get { return VCantidadEstanoKg; }
        set
        {
            if (value < 1) { MessageBox.Show("La cantidad debe ser mayor a 1 Kg"); }
            else { this.VCantidadEstanoKg = value; }
        }
    }
}
3 referencias
```

Ej0024

```
public class Proceso
{
    decimal VQCobre = 0;
    decimal VQEstano = 0;
    decimal VQBronce = 0;
    1 referencia
    public decimal SobranteCobre
    { get { return VQCobre; } }
    1 referencia
    public decimal SobranteEstano
    { get { return VQEstano; } }
    1 referencia
    public decimal BronceProducido
    { get { return VQBronce; } }
    1 referencia
    public void ProducirBronce(Entrada pEntrada)
    {
        if (pEntrada.CantidadCobreKg < 1 || pEntrada.CantidadEstanoKg < 1)
        { MessageBox.Show("Alguna o ambas cantidades de los metales de ingreso es menor a 1 !!!"); }
        else
        {
            this.VQCobre = pEntrada.CantidadCobreKg;
            this.VQEstano = pEntrada.CantidadEstanoKg;
            while (VQCobre >= 1)
            {
                if (VQEstano >= 9) { VQBronce++; VQCobre--; VQEstano -= 9; }
                else {break;}
            }
        }
    }
}
2 referencias
```

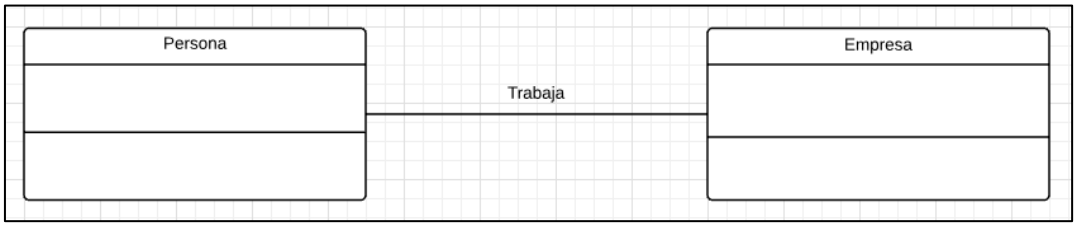
Ej0024

```
public class Salida
{
    1 referencia
    public void VerResultados(Proceso pProceso)
    {
        MessageBox.Show("El bronce producido es: " + pProceso.BronceProducido + " Kg\n" +
                        "Cobre sobrante: " + pProceso.SobranteCobre + " Kg\n" +
                        "Estaño sobrante: " + pProceso.SobranteEstano + " Kg");
    }
}
```

Ej0024

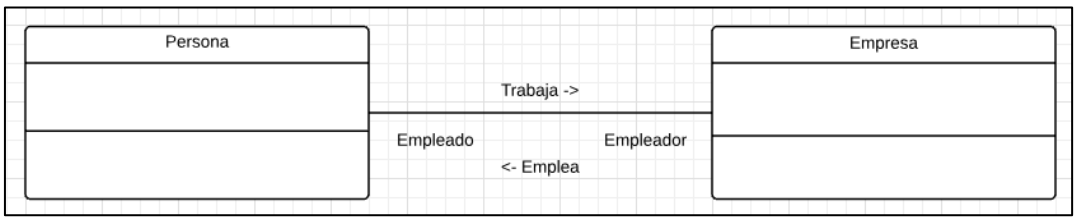
2. Asociación y Relación de Uso

La relación de **asociación** se da cuando las clases se conectan de forma conceptual. A diferencia de las anteriores relaciones, no observamos una relación todo-parte. La **asociación** surge de la propia relación con un verbo que la identifique.

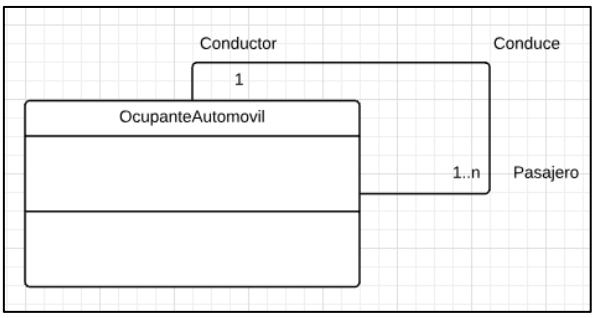


En una **asociación** se pueden establecer roles, como el rol de **empleador** y **empleado** en la relación precedente.

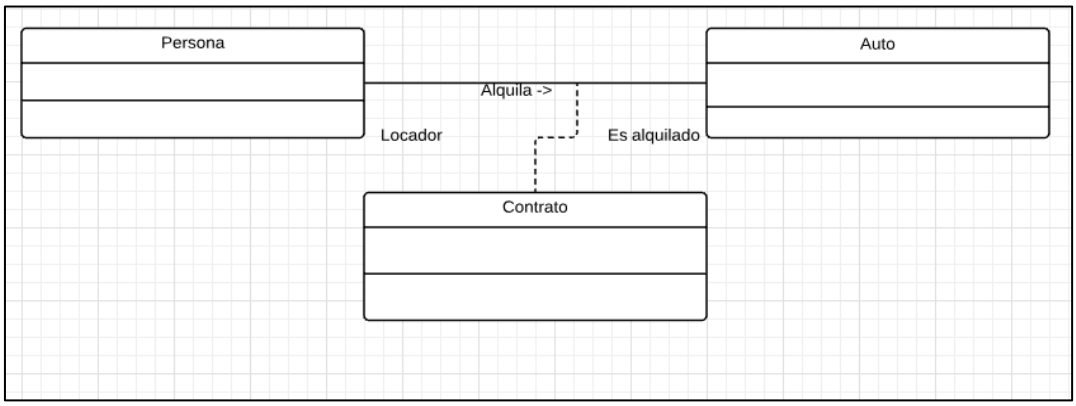
Se puede establecer una asociación bidireccional como se puede observar a continuación:



También se pueden producir asociaciones reflexivas o unarias. Esto se da cuando existen casos en los que una clase se puede relacionar consigo misma. Cuando una clase se puede relacionar consigo misma es porque puede adquirir diferentes roles.



Existen oportunidades que en las asociaciones nos encontramos con que necesitan atributos y/o métodos específicos. Esto se sustenta en la necesidad de manejar de forma idónea y más eficiente los elementos que surgen de dicha relación. A estas clases las denominamos **clases asociativas**. En el siguiente ejemplo la nueva clase **Contrato** contiene atributos y métodos específicos de la relación entre el **Locatario** y el **Locador**.



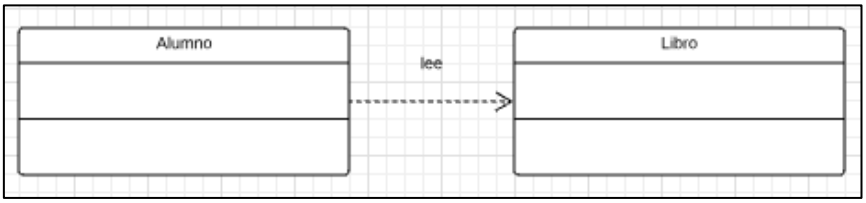
En algunas oportunidades es posible que se generen dudas para determinar si entre dos clases existe una relación de asociación o una de agregación.

Para facilitar esta decisión se pueden revisar algunos aspectos que facilitarán la tarea. En primer lugar la agregación determina relaciones de todo-parte, por lo cual siempre tendremos elementos que son "partes" del "todo". En general cuando se existe una asociación se detecta una conexión del tipo "usa un" o "tiene un".

En las relaciones de asociación se permite el conocimiento de ambas clases, es decir, puede existir la bidireccionalidad, sin embargo las relaciones de agregación son unidireccionales. El "todo" conoce a las "partes" pero no al revés. Otro aspecto a considerar es que la cardinalidad de la agregación es de 1..1 (uno a uno) o 1..n (uno a muchos), mientras que en la asociación puede ser de 1..1 (uno a uno), 1..n (uno a muchos) y n..n (muchos a muchos).

Para finalizar este punto definimos que es una **relación de uso**. Una **relación de uso** es una asociación refinada, donde se establece quien va a hacer uso de quien.

Por ejemplo si tenemos la clase **Alumno** y la clase **Libro**, y entre ellas existe una relación donde **Alumno** retira un libro para leer, entre ellas existe una **relación de uso**, o sea, es básicamente una asociación, pero claramente **Alumno** usa **Libro** para leerlo.



En el ejemplo **Ej0025** se puede observar como el método **Lee** recibe una referencia al libro que tiene que leer. Los siguientes fragmentos de código ejemplifican lo dicho.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1() {InitializeComponent(); }
    1 referencia
    private void Form1_Load(object sender, EventArgs e) { }
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        Alumno A = new Alumno(Interaction.InputBox("Ingrese el nombre del alumno: "));
        Libro L = new Libro(Interaction.InputBox("Ingrese el título del libro: "),
                            Interaction.InputBox("Ingrese el contenido del libro: "));
        MessageBox.Show("El Alumno " + A.Nombre + " ha leido el libro '" + L.Titulo + "'" +
            "\n\nCuyo contenido es: " + A.Lee(L));
    }
}
```

Ej0025

```
public class Alumno
{
    private string Vnombre;
    1 referencia
    public Alumno(string pNombre) {this.Vnombre = pNombre;}
    1 referencia
    public string Nombre { get { return this.Vnombre; } }
    1 referencia
    public string Lee(Libro pLibro) { return pLibro.Contenido;}
}
4 referencias
public class Libro
{
    private string Vcontenido;
    private string Vtitulo;
    1 referencia
    public Libro(string pTitulo, string pContenido)
    {this.Vcontenido = pContenido; this.Vtitulo = pTitulo;}
    1 referencia
    public string Contenido { get { return this.Vcontenido; } }
    1 referencia
    public string Titulo { get { return this.Vtitulo; } }
}
```

Ej0025

7. Elementos que determinan la calidad de una clase

Los elementos para determinar si una clase u objeto están bien diseñados son los siguientes:

- Acoplamiento
- Cohesión
- Suficiencia
- Compleción
- Ser Primitivo

Acoplamiento: Es la medida de la fuerza de la asociación establecida por una conexión entre una clase y otra. El acoplamiento alto en un sistema lo hace más complejo de mantener. Incrementar la complejidad del sistema produce que sea más difícil de comprender y corregir. El acoplamiento alto es igualmente nocivo a nivel de clases como de objetos. La herencia como mecanismo tiende a aumentar el acoplamiento, lo que determinará cierta rigidez estructural en el diseño. En numerosas oportunidades se puede reemplazar la herencia por mecanismos que generan menos acoplamiento y son más flexibles como el uso de interfaces, tipos genéricos etc.

Lo deseable en un diseño es que entre las clases el acoplamiento sea bajo o débil.

Cohesión: La cohesión mide el grado de conectividad entre los elementos internos de una clase u objeto. La cohesión funcional es un atributo deseable en una clase u objeto. Esta se evidencia cuando los elementos que componen la clase u objeto interactúan significativamente todos juntos para lograr un comportamiento bien delimitado. Es positivo cuando este tipo de cohesión es alta.

Suficiencia: Este atributo se da cuando la clase captura suficientes características de la abstracción, como para permitir una interacción significativa y eficiente.

Compleción: Se da cuando la interfaz de la clase captura todas las características significativas de la abstracción. Una clase es completa cuando su interfaz goza de niveles de usabilidad significativos. El concepto de usabilidad significativa se refiere a que se exponga todo lo necesario para acceder a las funcionalidades que la abstracción posee, pero evitar acciones de alto nivel que se puede obtener recombinando las de bajo nivel existentes.

Ser primitivo: Se refiere a las operaciones que se implantan solo accediendo a su representación interna. Las operaciones menos primitivas se construyen recombinando operaciones primitivas. Es deseable que las operaciones sean primitivas, pero sin comprometer la usabilidad general del objeto. Debe existir un compromiso entre que tan primitivas son las operaciones, ya que se constituyen

como operaciones altamente recombinantes, y la usabilidad que se le dará a quien utilice la clase u objeto en cuestión.

3. Relaciones entre Objetos: Enlace y Agregación

Si nos concentramos estrictamente en el aspecto dinámico del modelo donde operan los objetos, las relaciones que se generan entre las clases tienen su manifestación a nivel de objetos, pues estos son instancias de esas clases.

A nivel de objetos, decimos que dos objetos están enlazados cuando denotan una asociación específica por la cual uno puede solicitarle servicios al otro. Al objeto que solicita los servicios lo denominamos **objeto cliente** y a quien brinda los servicios **objeto servidor**. Para que un objeto A le pueda solicitar algo a un objeto B, A le envía un **mensaje** a B.

Cuando los objetos se enlazan, los mismos adoptan roles. Los roles que pueden adoptar son:

- Actor
- Servidor
- Agente

Actor: Un objeto se define como **Actor** cuando él puede operar sobre otros objetos, pero los demás no pueden operar sobre él. Los términos **objeto activo** y **Actor** se utilizan como sinónimos.

Servidor: Un objeto se define como **Servidor** cuando él no opera sobre otros objetos, pero los demás pueden operar sobre él.

Agente: Un objeto se define como **Agente** cuando él puede operar sobre otros objetos y los demás pueden operar sobre él.

Para que un objeto pueda enviarle un **mensaje** a otro, este último debe tener **visibilidad** a él.

Dados dos objetos A y B, para que A (**Actor**) le pueda enviar un **mensaje** a B (**Servidor**), B debe ser visible para A.

Esto puede ocurrir si de da algunas de las siguientes situaciones para el objeto B:

- El objeto B es global para el objeto A.
- El objeto B es un parámetro de alguna operación del objeto A.
- El objeto B es parte del objeto A.
- El objeto B es un objeto declarado localmente en alguna operación del objeto A.
-

Actividades asincrónicas

UAI

Universidad Abierta Interamericana

ACTIVIDAD INTEGRADORA NRO II

PROGRAMACIÓN ORIENTADA A OBJETOS

Profesor: DARIO CARDACCI

Nos solicitan crear un programa que permita administrar los inversores que compran y venden acciones para la empresa **Invest S.A.**
Los inversores son personas que se las identifica por un legajo que se genera en el sistema al momento de darlos de alta. Además, se solicita que el usuario ingrese el apellido, nombre y dni.
Los inversores invierten en acciones de empresas. Cada inversor puede tener inversiones en distintas acciones y de cada una de ellas poseer distintas cantidades.
Las acciones se identifican por un código compuesto por tres partes divididas por guiones. La primera parte son caracteres alfabéticos en mayúscula que identifican a la empresa. La segunda cuatro números que son de validación. La tercera una combinación de cuatro caracteres para un control interno, el primero y tercero son letras y el segundo y el curto son números (BGAL-4148-J4T3). Además, las acciones poseen una denominación, la cotización actual y la cantidad emitida, la cual nunca podrá ser superada por las compras de los inversores.
Como se mencionó anteriormente los inversores pueden comprar y vender acciones. La inversión de un inversor se calcula por cuanto dinero posee considerando las acciones que posee y su cotización actual.
Cada vez que una acción cambia su cotización, el inversor debe enterarse por medio de un evento que se desencadena en la acción que cambió su cotización. El evento lleva en su argumento personalizado la cotización actual.
También al realizar la ingeniería de requerimientos descubrimos que hay dos tipos de inversores. Un grupo está representado por los inversores comunes y otro por los inversores Premium. Los primeros cada vez que realizan una operación de compra o venta se les cobra una comisión del 1% sobre el total. Los Premium también pagan el 1% por compras y ventas hasta 20.000 pesos. Superada esa base sobre el resto abonan la mitad del porcentaje consignado anteriormente.

Nos solicitan que la GUI (interfaz gráfica del usuario) permita visualizar en grillas DataGridView:

1. La lista de todos los inversores (todos sus datos). (Grilla 1)
2. Las acciones que posea el inversor selecciondo en la Grilla 1 (todos los datos de las acción + cuantas acciones posee el inversor + el valor total de la inversión [total acciones * cotización]). (Grilla 2)
3. Todas las acciones (todos sus datos) en la que el inversor puede invertir. (Grilla 3)

La GUI debe tener botones para:

a. Agregar inversores y acciones.
b. Borrar inversores y acciones.

c. Modificar inversores y acciones.
d. Que un inversor pueda comprar acciones.
e. Que un inversor pueda vender acciones.

Nos solicitan:

Validar los datos ingresados.

Utilizar Try Catch y generar las excepciones personalizadas necesarias para informar sobre problemas de validación, imposibilidad de compras por no haber acciones disponibles, borrados o modificaciones de datos inexistentes etc.

Colocar constructores y destructores a las clases.

Utilizar en cada clase la Intefaz IDisposable.

Aplicar Herencia donde corresponda.

Aplicar Agregación y composición donde corresponda.

Aplicar Asociación donde corresponda.

Clonar donde corresponda.

Que los Accionistas se puedan ordenar de manera ascendente y descendente por (Legajo, Nombre, Apellido, DNI). Se deben utilizar interfaces.

Que las Acciones se puedan ordenar de manera ascendente y descendente por (Legajo, Nombre, Apellido, DNI). Se deben utilizar interfaces.

Que se pueda obtener el código de la acción seleccionada en la grilla 3 iterando la acción con un foreach de manera que en cada iteración retorne cada una de las partes que lo componen sin los guiones. Se debe utilizar interfaces.

La GUI debe tener un label donde se observe el total invertido por el inversor seleccionado en la grilla 1.

La GUI debe tener 4 labels donde se observe el total ganado por la empresa en concepto de comisiones cobradas por compras y ventas. La información que se observa en cada uno de ellos es:

a) Label1: El total recaudado por operaciones de los clientes comunes.
b) Label2: El total recaudado en concepto de comisiones por operaciones de los clientes premium por los ingresos correspondientes hasta 20.000.
c) Label3: El total recaudado en concepto de comisiones por las operaciones de los clientes premium por los ingresos correspondientes que superan los 20.000.
d) Labal 4: el total general percibido en concepto de comisiones.

Recursos a considerar para la resolución:

1. **Tema** Listas de programación 1. **Sugerencia**: Usar **List(of…)**
2. **Tema** Clses de vista.
3. **Tema** Clases, Instancias, Propiedades, Métodos, Constructores, Finalizadores, Eventos e Interfaces. **Sugerencia**: Ver los Orientador y la bibliografía recomendada.
4. **Tema** Try … Catch
5. **Tema** DatagridView. https://docs.microsoft.com/es-es/dotnet/framework/winforms/controls/datagridview-control-windows-forms