




Acceso a datos

Patrones



Generics

- Los genéricos introducen en .NET Framework el concepto de parámetros de tipo, lo que le permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declare y cree una instancia de la clase o el método
- Introducidos en la versión 2.0 de la biblioteca de clases .NET Framework
- Las clases y métodos genéricos combinan reusabilidad, seguridad de tipos y eficacia de una manera en que sus homólogos no genéricos no pueden.
- Los genéricos se usan frecuentemente con colecciones y los métodos que funcionan en ellas



Por ejemplo, al usar un parámetro de tipo genérico T puede escribir una clase única que otro código de cliente puede usar sin incurrir en el costo o riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing, como se muestra aquí:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Información general sobre los genéricos

- Use tipos genéricos para maximizar la reutilización del código, la seguridad de tipos y el rendimiento.
- El uso más común de los genéricos es crear clases de colección.
- La biblioteca de clases .NET Framework contiene varias clases de colección genéricas nuevas en el espacio de nombres [System.Collections.Generic](#). Estas se deberían usar siempre que sea posible en lugar de clases como [ArrayList](#) en el espacio de nombres [System.Collections](#).
- Puede crear sus propias interfaces, clases, métodos, eventos y delegados genéricos.
- Puede limitar las clases genéricas para habilitar el acceso a métodos en tipos de datos determinados.
- Puede obtener información sobre los tipos que se usan en un tipo de datos genérico en tiempo de ejecución mediante la reflexión

Patrón Unidad de Trabajo

- Este patrón tiene como objetivo tratar como una Unidad todos aquellos objetos nuevos, modificados o eliminados con respecto de una fuente de datos.
- Se utiliza para trabajar con un conjunto de objetos persistentes que deben tratarse como una "unidad" de trabajo, almacenándose en una base de datos de manera atómica.
- Este patrón es el encargado de hacer el seguimiento de todos aquellos objetos que son nuevos, y que por lo tanto deben guardarse en la base de datos, de todos los objetos que han sido modificados y que deben actualizarse en la base de datos y de todos los que han sido borrados y deben quitarse de la base de datos.

Unit of Work

```
registerNew(object)
registerDirty(object)
registerClean(object)
registerDeleted(object)
commit
rollback
```



Conclusiones

- Mantiene una lista de objetos afectados por una transaccion del negocio y coordina la escritura de cambios y la resolucion de problemas de concurrencia.
- El patrón UoW nos va a resultar muy útil a la hora de persistir un conjunto de acciones a ejecutar sobre la base de datos, evitando el exceso de conexiones contra la misma.



Patrón Repositorio

- El patrón repositorio se encarga de separar la lógica mediante la cual se accede a los datos almacenados para ser mapeados al modelo de mi negocio o dominio de aplicación, realizar las consultas a la base de datos o impactar nuestra entidad del modelo al origen de datos utilizado.
- En este sentido nos permite que la capa de negocio sea agnóstica en cierta forma de la tecnología de almacenamiento que se utilice.



Motivaciones para su uso

- Separar la lógica de negocio del acceso a base de datos, es decir evitar que la capa de negocio acceda directamente,
- lo cual nos puede llevar a:
 - ❑ Duplicar código.
 - ❑ Posibilidad de errores a la hora de desarrollar.
 - ❑ Capa de negocios fuertemente acoplada con la capa de acceso a datos.
 - ❑ Dificultad para realizar pruebas unitarias de la capa de negocio sin dependencias externas.

- Un repositorio es una clase que servirá como intermediario entre nuestra aplicación y nuestros datos, dicho con otras palabras, será una clase que nos ofrecerá una interfaz CRUD (Crear, Obtener, Actualizar, Borrar (Create, Read, Update, Delete)).

```
1 public interface IRepository<T> where T : EntityBase
2 {
3     T GetById(int id);
4     IEnumerable<T> List();
5     IEnumerable<T> List(Expression<Func<T, bool>> predicate);
6     void Add(T entity);
7     void Delete(T entity);
8     void Edit(T entity);
9 }
10
11 public abstract class EntityBase
12 {
13     public int Id { get; protected set; }
14 }
```



```
1 public class Repository<T> : IRepository<T> where T : EntityBase
2 {
3     private readonly ApplicationDbContext _dbContext;
4
5     public Repository(ApplicationDbContext dbContext)
6     {
7         _dbContext = dbContext;
8     }
9
10    public virtual T GetById(int id)
11    {
12        return _dbContext.Set<T>().Find(id);
13    }
14
15    public virtual IEnumerable<T> List()
16    {
17        return _dbContext.Set<T>().AsEnumerable();
18    }
19
20    public virtual IEnumerable<T> List(System.Linq.Expressions.Expression<Func<T, bool>> predicate)
21    {
22        return _dbContext.Set<T>()
23            .Where(predicate)
24            .AsEnumerable();
25    }
26
27    public void Insert(T entity)
28    {
29        _dbContext.Set<T>().Add(entity);
30        _dbContext.SaveChanges();
31    }
32
33    public void Update(T entity)
34    {
35        _dbContext.Entry(entity).State = EntityState.Modified;
36        _dbContext.SaveChanges();
37    }
38
39    public void Delete(T entity)
40    {
41        _dbContext.Set<T>().Remove(entity);
42        _dbContext.SaveChanges();
43    }
44 }
```