Patrones de acceso de una aplicación basada en Scrum para la gestión de proyectos de desarrollo de software

Ing. José Raúl López Morales
Estudiante de Maestría en Sistemas Computacionales programa PNPC
Tecnológico Nacional de México / IT de Acapulco
Acapulco Gro., México
jraul_lopz@hotmail.com

M.C. José Francisco Gazga Portillo
Docente de Maestría en Sistema Computacionales
Tecnológico Nacional de México / IT de Acapulco
Acapulco Gro., México
ita.gazga@gmail.com

M.T.I. Juan Miguel Hernández Bravo
Docente de Maestría en Sistema Computacionales
Tecnológico Nacional de México / IT de Acapulco
Acapulco Gro., México
jmhernan@yahoo.com

M.I.D.S. Alma Delia de Jesús Islao
Docente de Maestría en Sistema Computacionales
Tecnológico Nacional de México / IT de Acapulco
Acapulco Gro., México
alma.islao.ita@gmail.com

Ing. Rogelio Ramírez Silva
Estudiante de Maestría en Sistemas Computacionales programa PNPC
Tecnológico Nacional de México / IT de Acapulco
Acapulco Gro., México
rrasilva18@gmail.com

***Resumen*—En este artículo, se plasma el trabajo interdisciplinario de la Maestría en Sistemas Computacionales con apoyo del CONACyT, impartida en el Instituto Tecnológico de Acapulco. El artículo tiene por meta, presentar la implementación de los patrones de unidad de trabajo y repositorio en la capa de acceso a datos, permitiendo agilizar y facilitar la manipulación de los datos respecto de la persistencia de datos a la capa de negocio. Se hace mención a que este artículo forma parte de una serie de trabajos siendo el tercero de estos, el cual da seguimiento a un artículo previo titulado: Diseño de una herramienta informática basada en la metodología Scrum para la gestión del desarrollo de software, escrito por los mismos autores y que fue presentado y publicado en el congreso de Academia Journals Puebla 2019.**

***Palabras clave*—Patrón de diseño, ASP.NET Core, MVC, Repositorio y Unidad de trabajo.**

I. Introducción

Actualmente existen una gran variedad de tecnologías para desarrollar sistemas informáticos que implementan patrones de diseño que permiten agilizar y facilitar el proceso de desarrollo, lo cual es debido a la gran demanda que existe en el mercado, que las empresas requieren de algún tipo de software que demande la menor cantidad de tiempo en su construcción. Cuando se habla de escalar sistemas informáticos, el patrón MVC (Modelo-Vista-Controlador) es ideal para este tipo de escenarios, porque resulta más fácil codificar, depurar y probar algo (ya sea del modelo, vista o controlador) que tenga un solo trabajo ya que resulta complejo actualizar, probar y depurar código que tenga dependencias repartidas entre dos o más de estas tres áreas. Por ejemplo, la lógica de la interfaz de usuario tiende a cambiar con mayor frecuencia que la lógica de negocios. Si el código de presentación y la lógica de negocios se combinan en un solo objeto, un objeto que contenga lógica de negocios deberá modificarse cada vez que cambie la interfaz de usuario [6]. ASP.NET Core MVC es un marco de desarrollo de aplicaciones web de Microsoft que combina la eficiencia y el orden de la arquitectura de MVC, ideas y técnicas de desarrollo ágil y las mejores partes de la plataforma .NET [1]. Este marco de desarrollo es de código abierto y multiplataforma permitiendo desplegar aplicaciones en diferentes sistemas operativos. Un grupo de especialistas conformado por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides han catalogado un conjunto de patrones. Actualmente, su catálogo se considera una de las fuentes de información más populares sobre patrones de diseño. Dado que los patrones fueron documentados por los cuatro autores, los patrones se denominan patrones de diseño Gang of Four, o GoF. El catalogo GoF incluye 23 patrones de diseño. Los autores han colocado estos 23 patrones en tres categorías; patrones de creación, estructurales y de comportamiento [4]. ASP.NET Core MVC permite utilizar inyección de dependencia, la cual es una técnica que ayuda a crear aplicaciones flexibles y simplifica las pruebas unitarias [1]. Esta técnica permite inyectar servicios antes de que muestre una vista al usuario, una forma de inyectar servicios con esta técnica es a través del patrón *singleton* el cual dicta que solamente se puede crear una instancia de una clase, entonces cuando se crean aplicaciones web con ASP.NET Core MVC que accedan a una base de datos, es obligado crear un contexto (clase) donde se establezca una cadena de conexión a la base de datos. Al inyectar un servicio del contexto a través del patrón *singleton*, se permite mejorar el rendimiento de una aplicación web debido a que, si múltiples usuarios realizan transacciones,

Consejo Nacional de Ciencia y Tecnología

estos podrán acceder a la base de datos a través de un solo objeto del contexto. El patrón *singleton* pertenece a la categoría creacional del catálogo GoF.

A. *Patrones de arquitectura de aplicaciones empresariales*

Existe otro catálogo de patrones compilado por Martin Fowler. Este catálogo es llamado Patrones de Arquitectura de Aplicaciones Empresariales (en inglés Patterns of Enterprise Application Architecture ó P of EAA). Los patrones de ambos catálogos (GoF y P of EAA) se usan al construir sistemas de software. Sin embargo, los P de EAA están orientados a las aplicaciones empresariales [4]. Una aplicación empresarial es una colección de componentes que proporciona una funcionalidad empresarial que se puede utilizar internamente, externamente o con otras aplicaciones empresariales. Por ejemplo, gestión de pedidos, de inventarios o facturación [2]. Estas aplicaciones suelen ser complejas, altamente escalables y distribuidas. Los patrones del catálogo P of EAA están colocados en 10 categorías: Patrones de lógica de dominio, arquitectónicos de orígenes de datos, de comportamiento relacional de objetos, estructurales relacional de objetos, de mapeo de metadatos de objetos relacionales, de presentación web, de distribución, de concurrencia fuera de línea, de estado de sesión y de base. Las aplicaciones modernas están divididas en tres capas conceptuales: Acceso a datos, lógica de negocio y presentación o interfaz de usuario.

Los patrones unidad de trabajo y repositorio que pertenecen al catálogo P of EAA, permiten obtener la mejor forma de realizar transacciones con la base de datos. A continuación, se explican estos patrones.

B. *Patrón repositorio*

Emplear una capa de acceso a datos para realizar las operaciones Crear, Leer, Actualizar y Eliminar (en inglés Create, Read, Update y Delete, CRUD) es muy común en las aplicaciones del mundo real. Un componente de acceso a datos aísla los detalles de cómo las operaciones CRUD están tomando lugar en un sistema informático, por ejemplo, cuando se necesita realizar un conjunto de consultas de la misma base de datos desde varios lugares. Aunque un componente de acceso a datos encapsule las operaciones básicas de CRUD, no ofrecerá ayuda para evitar la duplicación de estas operaciones. Por lo tanto, se viene realizando el mismo código en varios lugares de una aplicación. En tales casos, se puede introducir una capa entre las clases de dominio y la capa de acceso a datos. Esta capa se encargará de las operaciones de encapsulación que se pueden reutilizar una y otra vez. Esta capa viene en forma de patrón repositorio. El patrón repositorio media entre la capa de acceso a datos y el resto del sistema. Además, lo hace proporcionando acceso de colección a los datos subyacentes. Una vez que se implementa el patrón de repositorio, el código de la capa de negocio no invocará el componente de acceso a datos directamente. En su lugar, invocará el repositorio para hacer el trabajo. El patrón repositorio ofrece una interfaz de recopilación al proporcionar métodos para agregar, modificar, eliminar y recuperar objetos de dominio.

C. *Patrón unidad de trabajo*

Una operación comercial involucra múltiples pasos mientras es requerida. La operación puede ser tratada como un solo lote o unidad. Se puede recurrir a las transacciones de la base de datos y realizar la tarea de la siguiente manera:

- Comenzar una transacción en la base de datos.
- Realizar todos los pasos de la operación uno por uno contra la base de datos.
- Comprometer o revertir la transacción.

Aunque este procedimiento parece sencillo, existe un problema: Se están realizando pasos individuales de la operación directamente en la base de datos. Entonces, si una operación involucra n pasos, está creado n operaciones de escritura en la base de datos. Una cantidad tan grande de pequeñas operaciones en la base de datos puede afectar el rendimiento general del sistema. Una mejor forma es capturar pasos individuales y luego enviarlos al motor de la base de datos de una sola vez y luego ejecutarlos en una única transacción, esa es la idea detrás del patrón unidad de trabajo. El patrón de unidad de trabajo realiza un seguimiento de la transacción comercial que se supone altera la base de datos de alguna manera. Una vez que finaliza la transacción comercial, los pasos se reproducen en la base de datos en una transacción para que la base de datos refleje los cambios deseados. Por lo tanto, el patrón de unidad de trabajo rastrea una transacción comercial y la traduce en una transacción de base de datos, en la que los pasos se ejecutan colectivamente como una sola unidad [4].

II. Objetivo general

El objetivo principal que se pretende lograr con la presente propuesta, consiste en desarrollar una herramienta informática que permita la gestión de proyectos de desarrollo de software que basan su construcción, en el empleo de la metodología Scrum, para ello, es importante la implementación de los patrones unidad de trabajo y repositorio en la capa de acceso, haciendo uso del lenguaje de programación C# con el marco de desarrollo ASP.NET Core MVC.

III. Implementación de los patrones repositorio y unidad de trabajo

A partir de esta sección, se presenta la implementación de los patrones repositorio y unidad de trabajo en la capa de acceso a datos de la herramienta, esta implementación es soportada por el diagrama de clases propuesto en el segundo artículo de la serie [3], en el cual se representó dichos patrones.

A. *Repositorio*

En este apartado se presenta algunos métodos implementados que proporciona el patrón repositorio al momento de utilizarlo. Las siguientes líneas de código implementan la clase genérica ***RepositorioGenerico*** la cual hereda la interfaz ***IRepositorioGenerico*** indicando a través de la palabra reservada ***where*** que ***T*** es una clase.

```
public class RepositorioGenerico<T> :
IRepositorioGenerico<T> where T : class
{
 internal PiScrumDbContext db = null;

 public
 RepositorioGenerico(PiScrumDbContext db)
 {
  this.db = db;
 }
}
```

Dentro de la clase ***RepositorioGenerico*** se crea un campo el cual es de tipo ***PiScrumDbContext*** que es una clase que representa el contexto para realizar operaciones en la base de datos. A continuación, se muestran las líneas de código que implementan el método para agregar (***AgregarAsin***) un registro en la base de datos.

```
public virtual async Task<T>
AgregarAsin(T entity)
{
   db.Set<T>().Add(entity);
   return entity;
}
```

El método ***AgregarAsin*** recibe como parámetro ***entity*** de tipo ***T***, el cual es proporcionado como argumento para el método ***Add*** donde a través del método ***Set*** se obtiene un ***DbSet*** (que representa la colección de todas las entidades en el contexto) de la entidad para que se pueda guardar la instancia de la entidad (clase). Los métodos ***Add*** y ***Set*** son proporcionados por ***Entity Framework*** el cual es un mapeador de objetos relacionales que permite a los desarrolladores de .NET trabajar con una base de datos utilizando objetos .NET. Las siguientes líneas de código permiten actualizar datos de un registro en la base de datos.

```
public virtual async Task<T>
ActualizarAsin(T entity, object key)
{
  if (entity == null)
    return null;
  T exist = await
   db.Set<T>().FindAsync(key);
  if (exist != null) {
     db.Entry(exist).CurrentValues.SetVal
     ues(entity);
  }
  return exist;
}
```

El método ***ActualizarAsin*** presentado, recibe como parámetros ***entity*** y ***key*** donde ***entity*** es de tipo ***T*** y ***key*** de tipo ***object***. En este método primero se realiza una consulta (***FindAsync***) en la entidad almacenada en ***entity*** donde se hace un filtrado utilizando el parámetro ***key***. La variable ***exist*** almacena la entidad retornada por la consulta, entonces la segunda condición se cumple y se hace la actualización a través del método ***Entry*** proporcionando como argumento la entidad retornada a la cual se va actualizar con la información almacenada en ***entity*** por el método ***SetValues***. Los métodos ***FindAsync***, ***Entry*** y ***SetValues*** también son proporcionados por ***Entity Framework***. A continuación, en las siguientes líneas se presenta el método para eliminar un registro de la base de datos.

```
public virtual async Task EliminarAsin(T
entity)
{
   db.Set<T>().Remove(entity);
}
```

El método ***EliminarAsin*** presentado, recibe como parámetro ***entity*** de tipo ***T*** donde almacena la información del registro a eliminar de una entidad específica. Para eliminar un registro simplemente se utiliza el método ***Remove*** donde se pasa como argumento ***entity*** y este obtiene un ***DbSet*** por el método Set para realizar la transacción en el contexto.

Como pueden observar en la firma de los métodos presentados tienen dos palabras reservadas ***async*** y ***Task***, esto hace que los métodos sean asíncronos, permitiendo que se puedan hacer múltiples transacciones en el contexto con una sola instancia de este.

B. Unidad de trabajo

En las siguientes líneas de código se muestra la parte esencial de la implementación del patrón unidad de trabajo.

```
public class UnidadDeTrabajo :
IUnidadDeTrabajo
{
  private readonly PiScrumDbContext db;

  public UnidadDeTrabajo(PiScrumDbContext
      db)
  {
    this.db = db;
  }

  private IRepositorioGenerico<Proyectos>
      Proyectos;

  private IRepositorioGenerico<
      HistoriasUsuario>HistoriasUsuario;

  private IRepositorioGenerico<Sprint>
      Sprint;

  private IRepositorioGenerico<Tareas>
      Tareas;

  private IRepositorioGenerico<
      SeguimientoSprint>
      SeguimientoSprint;

  public IRepositorioGenerico<Proyectos>
      ProyectosRepositorio
  {
    get
    {
```

```
        return Proyectos = Proyectos ?? new
        RepositorioGenerico<Proyectos>(db);
      }
    }

    public IRepositorioGenerico<
        HistoriasUsuario>
        HistoriasUsuarioRepositorio
    {
     get
      {
        return HistoriasUsuario =
        HistoriasUsuario ?? new
        RepositorioGenerico<HistoriasUsuari
        o>(db);
      }
    }

    public IRepositorioGenerico<Sprint>
        SprintRepositorio
    {
     get
     {
       return Sprint = Sprint ?? new
        RepositorioGenerico<Sprint>(db);
     }
    }

    public IRepositorioGenerico<Tareas>
        TareasRepositorio
    {
     get
     {
       return Tareas = Tareas ?? new
        RepositorioGenerico<Tareas>(db);
     }
    }

    public IRepositorioGenerico<
        SeguimientoSprint>
        SeguimientoSprintRepositorio
    {
     get
     {
       return SeguimientoSprint =
        SeguimientoSprint ?? new
        RepositorioGenerico<SeguimientoSpri
        nt>(db);
     }
    }


    public async Task<int> SaveAsync()
    {
      return await db.SaveChangesAsync();
    }
}
```

La clase ***UnidadDeTrabajo*** tiene el propósito de asegurar que, usando varios repositorios, compartan un mismo contexto de base de datos (variable ***db***). De tal forma que, cuando se complete una unidad de trabajo, se pueda invocar al método ***SaveChanges*** en esa instancia del contexto, asegurando que todos los cambios relacionados se coordinen. Por ello, la clase necesita el método ***SaveAsync*** y una propiedad para cada repositorio. Las propiedades del repositorio retornan una instancia de repositorio de la cual se han creado instancias con la misma instancia de contexto de base de datos que las demás instancias de repositorio. En el código anterior, se crean variables de clase para el contexto de la base de datos y para cada repositorio (variable ***SeguimientoSprint***). En el caso de la variable ***db***, se crea una instancia de un nuevo contexto.

Las propiedades del repositorio comprueban si el repositorio existe. De otra forma, se crea una nueva instancia de este, pasando la instancia de contexto (***db***). Lo cual da como resultado, cada uno de los repositorios que comparten la misma instancia de contexto.

IV. Resultados

Este apartado se presenta el uso de los patrones repositorio y unidad de trabajo en la lógica de negocio. El siguiente código es un fragmento del método que realiza el seguimiento del sprint de un proyecto en la herramienta.

```
await
unidadDeTrabajo.SeguimientoSprintReposito
rio.AgregarAsin(Modelo);
//guardando cambios en las entidades de
tareas e historias de usuario
if (ModeloT.EstimacionEsfuerzoRestante ==
0)
 ModeloT.Estatus = "FINALIZADO";
else
 ModeloT.Estatus = "PROCESO";

if (ModeloH.EstimacionEsfuerzoRestante ==
0)
 ModeloH.Estatus = "FINALIZADO";
else
 ModeloH.Estatus = "PROCESO";
```

```
await
unidadDeTrabajo.TareasRepositorio.Actuali
zarAsin(ModeloT,ModeloT.IdTarea);
```

```
await
unidadDeTrabajo.HistoriasUsuarioRepositor
io.ActualizarAsin(ModeloH,ModeloH.IdHisto
riaUsuario);
```

```
await unidadDeTrabajo.SaveAsync();
```

Del código anterior, las líneas con fondo gris, son donde se implementa el patrón de repositorio y unidad de trabajo. La primera línea en negrita se utiliza el método Agregar a través de la propiedad ***SeguimientoSprintRepositorio*** del repositorio

perteneciente a la clase ***UnidadDeTrabajo*** que implementa el patrón unidad de trabajo. El método Agregar requiere un modelo el cual es una clase de tipo ***SeguimientoSprint***. De esta misma forma se utiliza para actualizar una tarea y una historia de usuario. Cuando se realizan las transacciones comerciales agregar y actualizar se guardan cambios en la base de datos con las instancias de las entidades ***Seguimiento sprint, Historias Usuario*** y ***Tareas***. Estos cambios se hacen con el método ***SaveAsync*** a través del patrón ***unidad de trabajo*** como se muestra en el código presentado anteriormente. De esta misma forma se utiliza para actualizar una tarea y una historia de usuario. Cuando se realizan las transacciones comerciales agregar y actualizar se guardan cambios en la base de datos con las instancias de las entidades ***Seguimiento sprint***, ***Historias Usuario*** y ***Tareas***. Estos cambios se hacen con el método ***SaveAsync*** a través del patrón ***unidad de trabajo*** como se muestra en el código presentado anteriormente.

Cuando se ejecuta el método ***AgregarAsin***, **Entity framework** traduce esta instrucción en una instrucción SQL para que el gestor de base de datos pueda interpretarlo y realizar la transacción. En la figura 1 se observa la traducción del método ***AgregarAsin*** en SQL.

```
INSERT INTO "SeguimientoSprint" ("IdSeguimientoSprint", "FechaRegistro", "IdSprint", "IdTarea",
VALUES (@p2, @p3, @p4, @p5, @p6);
```

Fig 1. Traducción del método ***AgregarAsin*** a SQL.

De la misma forma, cuando se ejecuta el método ***ActualizarAsin*** este se traduce a SQL como se observa en la figura 2.

```
UPDATE "Tareas" SET "EstimacionEsfuerzoRestante" = @p7
WHERE "IdTarea" = @p8;
UPDATE "HistoriasUsuario" SET "Estatus" = @p0
WHERE "IdHistoriaUsuario" = @p1;
```

Fig 2. Traducción del método ***ActualizarAsin*** a SQL.

V. Comentarios finales

A. *Conclusiones*

En este artículo, se presentó la implementación de los patrones repositorio y unidad de trabajo en la capa de acceso permitiendo mejorar el rendimiento al momento de realizar transacciones en la base de datos, también influye el hecho de que los métodos contenidos en el patrón son asíncronos, resolviendo así, el problema de concurrencia. La forma de tratar una transacción comercial el patrón unidad de trabajo, es muy similar al mecanismo de las instrucciones commit y rollback de SQL, es decir, si en una transacción con n cambios falla uno ellos entonces ninguno se refleja en la base de datos esto asegura la integridad de la base de datos. La implementación de estos patrones en la capa de acceso no afecta al momento de cambiar de gestor de base de datos ya que estos patrones como se expuso, el patrón repositorio tiene la función de crear una capa de abstracción entre la capa de acceso a datos y la lógica de negocio y el patrón unidad de trabajo permite manejar transacciones durante la manipulación de datos utilizando los métodos implementados en el patrón repositorio, es decir, el patrón unidad de trabajo solamente necesita de un contexto sin importar el origen.

B. *Trabajos a futuro*

Este artículo forma parte de una serie de trabajos que reflejan el progreso del proyecto de tesis de maestría, donde el trabajo a futuro claramente debe abordar la conclusión de la herramienta donde se expondrá, el desarrollo de la aplicación propuesta, haciendo uso de la implementación de los patrones repositorio y unidad de trabajo presentada en este trabajo y haciendo uso de las herramientas de desarrollo de software, implementando la arquitectura propuesta en el primer artículo [5].

Referencias

[1] A. Freeman, Pro ASP.NET Core MVC 2, Londres: Apress, 2017.

[2] IBM, «Aplicaciones empresariales,» IBM Corp., 2017. [En línea]. Available: https://www.ibm.com/support/knowledgecenter/es/SSPLFC_7.3.0/com.ibm.taddm.doc_7.3/UserGuide/c_cmdb_business_apps.html. [Último acceso: 22 Febrero 2020].

[3] J. F. Gazga Portillo, J. R. López Morales, J. M. Hernández Bravo y A. D. de Jesús Islao, «Diseño de una herramienta informática basadaen en la metodología Scrum para la gestión del desarrollo de software,» *Academia Journals*, pp. 789-794, 2019.

[4] B. Joshi, Beginning SOLID Principles and Design Patterns for ASP.NET Developers, Thane: Apress, 2016.

[5] J. R. López Morales, J. F. Gazga Portillo, J. M. Hernández Bravo y A. D. de Jesús Islao, «Propuesta de una herramienta basada en la métodología Scrum para la gestión del desarrollo de software,» vol. 11, nº 2, 2019.

[6] S. Smith, «Información general de ASP.NET Core MVC,» Microsoft, 28 01 2020. [En línea]. Available: https://docs.microsoft.com/es-es/aspnet/core/mvc/overview?view=aspnetcore-3.1. [Último acceso: 19 02 2020].