Aplicación bien diseñada

Publicado el 27 de julio de 2021

Esta publicación ayuda a comprender el proceso de desarrollo de aplicaciones bien diseñadas y la arquitectura de un proyecto de software desde los requisitos hasta una arquitectura en capas desacoplada completamente funcional.

No se centrará en la parte teórica del análisis comercial (como historias de usuarios o casos de uso) y tampoco desviará la atención a la creación de una aplicación web para evitar tener que lidiar con componentes de la interfaz de usuario (como HTML, JavaScript, CSS, etcétera). Esta publicación trabajará en los conceptos básicos de la programación orientada a objetos, como implementar varios patrones de diseño y usar varios estilos y principios arquitectónicos en un proyecto que resulte en código desacoplado y fácil de entender.

El proyecto resultante tendrá como objetivo crear una plantilla bien formateada con arquitectura desacoplada que se pueda implementar rápidamente en cualquier proyecto nuevo.

# Requisitos – ABC Pet Shop

“ABC Pet Shop” es una cadena de tiendas de mascotas ubicada en Sydney y Melbourne. La tienda de mascotas requiere un sistema de gestión de clientes simple para sus puntos de venta minoristas con las siguientes características (en una primera fase).

## Fase 1 – requisitos

1. La aplicación requerirá 5 campos: nombre completo, número de teléfono, monto de la factura, fecha de la factura y dirección del cliente.
2. Hay dos tipos de clientes que vienen a la tienda de mascotas: 1) cliente; 2) visitante.

* Un cliente es una persona que viene y compra cosas en una tienda y está involucrada en una transacción financiera. Todos los campos mencionados en 1 son obligatorios para los clientes.
* Un visitante es una persona que viene a la tienda de mascotas para hacer una consulta, pero no compra nada en ese momento. Solo un nombre completo y un número de teléfono son obligatorios para un visitante.
* El sistema debería poder agregar tipos de clientes si es necesario en el futuro

1. El sistema debe ser capaz de:

* Agregar nuevas reglas de validación sin problemas y estas reglas de validación deberán ser flexibles y reutilizables por el sistema.
* El sistema debe tener la capacidad de mostrar, agregar, actualizar y eliminar datos de clientes.
* Por lo pronto, el sistema deberá usar el servidor SQL y ADO.net como capa de datos y en los próximos meses deberá migrar a Entity Framework. La migración debe ser perfecta sin muchos cambios a lo largo del sistema.
* El sistema debe tener la capacidad de cancelar cualquier modificación realizada en la pantalla, lo que significa que, si estamos editando un registro y cambiando valores, deberíamos poder volver a los valores anteriores.

Estos son los requisitos iniciales para la Fase 1 del proyecto. Ahora, para continuar con estos requisitos, es esencial comprender algunos conceptos básicos de los principios orientados a objetos (OOP) para identificar las clases.

## ¿Qué es una Entidad?

“Una entidad es una cosa con existencia distinta e independiente”. Las entidades son cosas que vemos en el mundo real como sustantivos (por ejemplo, persona, lugar o cosas). En una perspectiva orientada a objetos, un objeto de entidad representa un objeto en el dominio del problema del mundo real. Entonces, en esta publicación, el objeto o las entidades se utilizarán como sinónimos.

# Pasos de desarrollo de una aplicación bien diseñada

## Paso 1 🡪 Identificar Entidades u Objetos de su sistema

El software se desarrolla principalmente para resolver problemas del mundo real y proporcionar automatización a escenarios y personas del mundo real. Un código de software debe replicar entidades del mundo real en sus objetos. Por ejemplo: un proveedor que suministra bienes en el mundo real. Entonces el código de software debe tener una entidad llamada Proveedor. Para identificar entidades/objetos necesitamos identificar sustantivos a partir de una especificación funcional o una historia de usuario. Los sustantivos se convierten en entidades, un verbo se convierte en acciones para las entidades y los pronombres se convierten en las propiedades y el comportamiento de esas entidades.

Entre los requisitos existen los siguientes sustantivos: Sydney, Melbourne, cliente, visitante, tienda de mascotas.

Existe una práctica común utilizada por los arquitectos de seguir sustantivos, verbos y pronombres. Pero un arquitecto debe tener cuidado de no elegir sustantivos y verbos no deseados que no se relacionen con el sistema. En la lista anterior de sustantivos, Sydney y Melbourne son ciudades y no tienen conexión directa con otros requisitos del sistema. Los únicos sustantivos útiles son cliente y visitante, que es donde el requisito solicita realizar acciones como validar, agregar, actualizar y eliminar.

### ¿Qué es una clase?

En OOP, una clase es un modelo (o plantilla) para crear objetos. La clase actúa como una plantilla que define la naturaleza de un objeto futuro (que debe contener el nombre, las acciones y las propiedades del objeto).

Diagrama

Descripción generada automáticamente

### Procesos trifásicos de POO

La POO es un proceso de tres fases, a saber:

1. **Creación de plantillas:** Creación de clases y lógica de escritura en esas clases.
2. **Instanciar:** Cree entidades/objetos de estas clases y tráigalos a vivir en la RAM/computadora.
3. **Ejecutar:** Interactuar con estos objetos para lograr la funcionalidad del software.

Un arquitecto de software/desarrollador de software encontrará problemas de diseño comunes/recurrentes en estas tres fases. Los patrones de diseño tienen soluciones para esos problemas.

Los patrones de diseño se dividen en tres categorías que cubren estas fases:

|  |  |
| --- | --- |
| **Fase POO** | **Categoría de Patrón de Diseño** |
| Creación de plantillas/clases | Estructural |
| Problemas de instanciación | Creacional |
| Problemas de tiempo de ejecución | De comportamiento |

Diagrama

Descripción generada automáticamente

Cree una clase de cliente con propiedades y acciones (métodos) en Visual Studio agregando una biblioteca de clases llamada "CustomerLibrary" en la solución "ABCPetShop":

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

## Paso 2 🡪 Identificar la relación entre dos entidades

En el mundo real estas entidades tienen principalmente dos relaciones “ES-UN” y “TIENE-UN”. Por ejemplo, la relación entre padre e hijo es una relación tipo "ES-UN": podemos decir "padre es hijo". Pero cuando comparamos padre con un automóvil o hijo con un automóvil, la relación cambia a una de tipo "TIENE-UN": "el padre tiene un automóvil” y “el hijo tiene un automóvil”.

En el requisito: “un visitante **es un** tipo de cliente con menos atributos”, la clase Visitante puede heredar las propiedades de la clase Cliente. La herencia es uno de los cuatro conceptos básicos de la POO (herencia, abstracción, encapsulación y polimorfismo).

Diagrama

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Una de las acciones que tiene que hacer el objeto Cliente es validar: tiene que asegurarse de que todos los atributos se completen para el cliente y FullName y PhoneNumber se completen para el Visitante. Por lo tanto, los métodos de validación deben ser diferentes. Eso significa que debemos asegurarnos de que el método Validate se pueda anular (make es "virtual"):

Texto

Descripción generada automáticamente

Hay un requisito más para que el sistema pueda agregar un nuevo tipo de cliente en el futuro. Esto implica que Cliente no es padre de Visitante sino un tipo de Cliente (sic) por lo que el padre es otra cosa del tipo cliente, visitante y cualquier otro tipo de cliente agregado. Llamémoslo una clase base (CustomerBase) de la cual podemos derivar clases de Cliente, Visitante y cualquier otro tipo de cliente.

Diagrama

Descripción generada automáticamente

CustomerBase es una clase incompleta con propiedades y métodos que el Cliente y el Visitante pueden derivar y usar según sus necesidades.

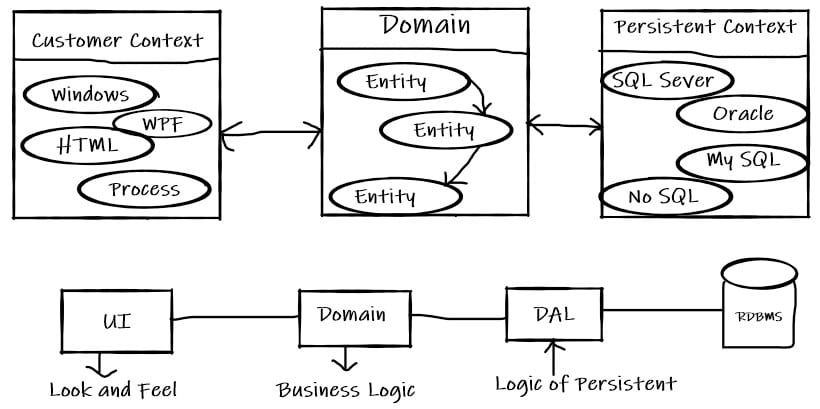
Texto

Descripción generada automáticamente

## Paso 3 🡪 Diseño de la arquitectura de la aplicación

En el proceso de desarrollo de aplicaciones bien diseñadas, hemos identificado las entidades (clases de dominio). Pero las preguntas aquí son "¿qué harán estas clases?" y "¿cómo funcionarán estas clases en el sistema?"

1. ¿Quién va a invocar estas clases (definir la interfaz de usuario o el contexto del cliente) para que funcionen?
2. Una vez que el trabajo está terminado por estas entidades, ¿quiénes van a almacenar el trabajo realizado por las clases (definir almacén de datos o contexto persistente).



La capa de presentación define la interfaz de usuario (UI) para invocar la capa comercial que tiene entidades/clases que representan objetos comerciales del mundo real (dominio) y la capa de datos almacenará las entidades.

### Arquitectura de tres capas

Gestionar 3 capas (IU, dominio, DAL) significa que "el cambio en una capa puede tener un impacto en 3 lugares o más". Por lo tanto, se necesita un diseño de arquitectura desacoplado para evitar el conflicto entre estas 3 capas.

La aplicación Windows Forms para UI, la biblioteca de clases para CusotmerLibrary (dominio) y la biblioteca de clases para DataAccess (DAL) se agregan al proyecto para crear una arquitectura de 3 capas:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

NOTA 🡪 Hay un término llamado Arquitectura de tres niveles que se usa principalmente de manera intercambiable con Arquitectura de tres capas. Existe la confusión de que estos dos son lo mismo, pero hay una gran diferencia entre los dos. La arquitectura de tres capas tiene tres capas lógicas separadas, pero está alojada en la misma máquina física; mientras que en la arquitectura de tres niveles las tres capas lógicas están alojadas en 3 máquinas diferentes.

## Paso 4 🡪 Diseño de interfaz

(Todos los nombres requeridos para las herramientas como cuadro de texto, botones de comando, etc. se dan en consecuencia):

Interfaz de usuario gráfica

Descripción generada automáticamente

El tipo de cliente separa al cliente del visitante y, en el futuro, si se agregan tipos adicionales, se enumerarán en consecuencia y la interfaz de usuario hace referencia a la biblioteca del cliente (capa de dominio).

using System;

using System.Windows.Forms;

using CustomerLibrary;

namespace PetShop

{

public partial class frmCustomer : Form

{

public frmCustomer()

{

InitializeComponent();

}

private void cmdAdd\_Click(object sender, EventArgs e)

{

CustomerBase custbase = null;

if (cboCustType.SelectedIndex==0)

{

custbase = new Visitor();

}

else

{

custbase = new Customer();

}

custbase.FullName = txtFullName.Text;

custbase.Address = txtAddress.Text;

custbase.PhoneNumber = txtPhoneNo.Text;

custbase.BillDate = dtBillDate.Value;

custbase.BillAmount = Convert.ToDecimal(txtBillAmount.Text);

}

}

}

El problema principal en el código anterior: ¿Qué sucederá si agregamos un tipo de cliente más? La interfaz de usuario tiene que cambiar cada vez que se agrega un nuevo tipo de cliente. Este es un problema de diseño que viola los principios de diseño de SOLID.

SOLID es un acrónimo de los primeros cinco principios de diseño orientado a objetos (DOO) de Robert C. Martin (tío Bob):

* S – Principio de responsabilidad única (SRP)
* O – Principio abierto-cerrado
* L – Principio de sustitución de Liskov
* I – Principio de segregación de interfaces
* D – Principio de Inversión de Dependencia

El Principio de Responsabilidad Única (SRP) establece que:

Una clase debe tener una y solo una razón para cambiar, lo que significa que la clase debe tener un solo trabajo. El sinónimo de SRP es Separación de Preocupaciones (SoC) que establece que una clase debe hacer solo sus preocupaciones y cualquier tarea no deseada (relacionada) debe trasladarse a otra clase.

En el código anterior, se viola el primer principio SOLID de SRP, la tarea de UI es recibir entradas y mostrar salidas, sin embargo, aquí tiene la tarea de crear nuevos objetos de Cliente, que no es su trabajo.

Interfaz de usuario gráfica, Diagrama, Aplicación

Descripción generada automáticamente

## Paso 5 🡪 Desacoplamiento de las clases: uso de interfaces

Para desacoplar la interfaz de usuario con las clases de clientes, la interfaz de usuario debe ver los tipos de clientes de forma abstracta (separada) en lugar de tratar con clases concretas.

La abstracción es uno de los conceptos clave de la programación orientada a objetos (POO) cuyo objetivo principal es ocultar detalles innecesarios del código solo para mostrar lo que es necesario.

La interfaz ayuda a implementar la abstracción mediante la creación de definiciones puras donde la interfaz de usuario apuntará a no preocuparse por las clases implementadas en el back-end. Se agrega una biblioteca de clases de interfaz (IcustomerLib) para lograr la abstracción.

