Universidad Veracruzana

Desarrollo de software

Estándar de codificación

Equipo 4

Miriam Ramírez Zárate

Ares Judda Rivera Soto

Alejandro Sánchez Marín

Paloma Osiris Báez Lara

29/03/2024

LIS

Contenido

Introducción

Las prácticas recomendadas en programación abarcan un conjunto de métodos, conceptos y normas empleados para crear código de excelente calidad y que pueda ser mantenido con facilidad. Estas pautas abordan aspectos clave como la legibilidad, el rendimiento, la capacidad de crecimiento, la seguridad y la facilidad de mantenimiento del código.

La importancia de aplicar prácticas óptimas en programación es diversa:

1. Simplifican la comprensión del código: Al utilizar convenciones y estructuras comunes y bien definidas, se vuelve más fácil para otros programadores entender el código escrito.
2. Mejoran la calidad del código: Las prácticas de programación adecuadas ayudan a prevenir errores y problemas, lo que resulta en un software más sólido y estable.
3. Incrementan la eficiencia: Al seguir prácticas como la optimización del código y la reutilización de componentes, se logra aumentar la velocidad de ejecución y reducir los recursos necesarios.
4. Promueven la escalabilidad: Al emplear patrones de diseño y arquitecturas bien establecidas, es posible crear software escalable, capaz de expandirse y evolucionar con el tiempo.
5. Garantizan la seguridad: Las prácticas adecuadas en programación contribuyen a prevenir vulnerabilidades y ataques maliciosos. En resumen, las buenas prácticas en programación son esenciales para producir software de alta calidad, que sea sencillo de mantener y que tenga la capacidad de crecer. Esto se traduce en un software más eficiente y seguro.
6. Detección temprana de fallas: Al buscar cumplir con los estándares es más sencillo detectar posibles errores desde la revisión de código, evitando que esos problemas lleguen a producción.

Propósito

El propósito principal es instaurar la norma que se seguirá a lo largo de todo el proceso de desarrollo del proyecto relacionado con los fundamentos de construcción de software en el lenguaje C#. El objetivo es dirigirse hacia la creación de un programa que combine tanto la excelencia como la sofisticación.

Para lograr este fin, se aplicarán las siguientes pautas:

1. Uniformidad en la estructura del código: Se mantendrá una coherencia en la estructura del código, empleando una indentación clara y constante. Además, se respetarán las convenciones de nomenclatura apropiadas, utilizando nombres descriptivos para variables, métodos y clases.
2. Modularidad y cohesión: El código se organizará en módulos lógicos y cohesionados, cada uno con una responsabilidad única y bien definida. Esto facilitará la comprensión y el mantenimiento del programa.
3. Incorporación de comentarios: Se incluirán comentarios claros y concisos en el código para explicar la funcionalidad y la lógica detrás de las implementaciones.
4. Prevención de duplicación de código: Se fomentará la reutilización de código mediante enfoques como la creación de funciones y clases genéricas, así como la

utilización de patrones de diseño. Esto reducirá la redundancia y mejorará la capacidad de mantenimiento del código.

5. Pruebas unitarias: Se implementarán pruebas unitarias para confirmar el correcto funcionamiento de cada componente y detectar posibles errores en etapas tempranas. Esto garantizará la calidad y la robustez del programa.
6. Manejo adecuado de excepciones: Se aplicará un manejo cuidadoso de las excepciones para asegurar que el programa sea resistente y no se vea interrumpido por fallos no controlados.

En resumen, siguiendo estas normas y prácticas de desarrollo, nuestro propósito es crear un sistema de gestión para la cafetería Komalli en C# que refleje distinción y sofisticación. Esto se traducirá en un código bien estructurado, de fácil comprensión, mantenible y eficiente.

Reglas o convención de nombres

Métodos

Los métodos tendrán nombres descriptivos usando verbos en infinitivo de acuerdo con lo que se realizará, así como se utilizará el estilo de escritura de PascalCase. No se permiten caracteres especiales.

Uso correcto:

```
public string CreateRoom(string gamertag) {

}
```

Uso no correcto:

```
public string generateId() {

}
```

Variables

Las variables se nombrarán usando "snake_case_with_underscore_prefix" si se trata de variables privadas, las públicas se hará llamándolas con la primera letra en mayúscula y para esto se hará usando sustantivos y palabras descriptivas de acuerdo con el uso que tendrán. Las excepciones son los ciclos for los cuales está permitido el uso de i, j, k, l siempre y cuando se siga ese orden si es que se anida más de un ciclo.

Uso correcto:

```
public int IdUser { get; set;

private string _gamertag;

for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        for (int k = 0; k < 10; k++) {
            for (int l = 0; l < 10; l++) {
```

```
            }
        }
    }
}
```

Uso no correcto:

```
private string Report;

public string _report;


for (int j = 0; j < 10; j++) {
      for (int i = 0; i < 10; i++) {

       }
}
```

Clases

Las clases estarán definidas usando el CamelCase y tendrán nombres descriptivos usando sustantivos. Debe evitarse el uso de acrónimos o abreviaturas, salvo en aquellos casos en los que dicha abreviatura sea más utilizada que la palabra que representa (URL, HTTP, etc.).

Uso correcto:

```
public class Friend {

}
```

Uso no correcto:

```
public class getStudents {

}
```

User controls

Los user controls estarán definidas usando el CamelCase y tendrán nombres descriptivos usando sustantivos. Debe evitarse el uso de acrónimos o abreviaturas, salvo en aquellos casos en los que dicha abreviatura sea más utilizada que la palabra que representa (URL, HTTP, etc.).

Uso correcto:

```
Public partial class FriendList : UserControl{

}
```

Uso no correcto:

```
public partial class playersgame : UserControl{

}
```

Constantes

Todas las contantes se escribirán en mayúsculas, de ser frases nominales se separarán las palabras entre si con guiones bajos (_).

Uso correcto:

```
const string CHARACTERS = "0123456789";
```

Uso no correcto:

```
const string image_profile = "/Komalli;component/Resources/Images/Profile.png";
```

Nombres de componentes

Los componentes se nombrarán comenzando en minúscula y todos tendrán el prefijo del componente que se trate y seguido será algo descriptivo de lo que realice.

Uso correcto:

```
x:Name="txbChat"
```

Uso no correcto:

```
x:Name="textBoxChat"
```

Control	Prefijo
Label	lbl
TextBox	txt
DataGrid	dtg
Button	btn
ImageButton	Imb
Hyperlink	hlk
DropDownList	ddl
ListBox	lst
DataList	dtl
Repeater	rep
Checkbox	chk
CheckboxList	cbl
RadioButton	rbt
RadioButtonlist	rbl
Image	img
Panel	pan
PlaceHolder	phd
Table	tbl
Validators	val

Stack panel – stp

Text Block – txb

Metodos de pruebas

Los métodos de prueba seguirán la convención CamelCase y se denominarán según el nombre del método que se está probando, seguido por un guion bajo y el tipo de prueba, ya sea "Successful," "Failed" o "Exception." En caso de que se realicen múltiples pruebas del mismo tipo en un método específico, después del tipo de prueba se especificará, mediante CamelCase, el número correspondiente (por ejemplo, "SuccessfulOne"). En situaciones en las que solo se realice una prueba de un tipo específico, no será necesario indicar el número de prueba.

Uso Correcto:

```
public void AddUserAccount_SucessfullOne () {
```

```
}
public void AddUserAccount_SucessfullTwo() {
}
public void UpdateInfoUser_Failed ( ){
}
public void ValidateUser_Exception () {
}
```

Uso no correcto:

```
public void AddUserAccount_Sucessfull2() {
}
public void UpdateInfoUser-failed ( ){
}
```

Controladores

Los controladores seguirán la convención de CamelCase y se denominaran según el tipo de evento que desencadenan, seguido de un nombre descriptivo sobre la función que realizan.

Uso correcto:

```
private void ClickPlay (object sender, RoutedEventArgs e){
}
```

Uso no correcto:

```
private void Options  (object sender, RoutedEventArgs e){
}
```

Organización de archivos

Se usará el patrón de diseño MVC, por ello se definirá 3 capas llamadas Model, View y Controller.

Dentro de la capa de Model se especificará las clases que hagan referencia a la estructura de los datos y la lógica del negocio. En la carpeta de View estará todo aquel archivo XAML que contenga el diseño de las interfaces de usuario. Y, por último, en la capa de Controller contendrá la lógica y manejará las interacciones del usuario.

Los nombres de los archivos y clases que se definan en cada capa serán simples, es decir, serán sustantivos y el nombre será descriptivo según lo que se realice.

Definición de métodos, clases y variables

Las clases y métodos serán de acceso público, sin embargo, las variables serán definidas como privadas, usando la estructura de nombrado "snake_case_with_underscore_prefix", para el uso de sus getters y setters los cuales serán en forma de propiedades, serán públicos y seguirán la estructura de nombrado CamelCase.

Uso correcto:

```
private string _gamertag;
public string Gamertag {
    get { return _gamertag; }
    set {
        if (_gamertag != value) {
            _gamertag = value;
            OnPropertyChanged(nameof(Gamertag));
        }
    }
}
```

Uso no correcto:

```
private string gamertag;

public string _Gamertag {
    get { return gamertag; }
    set {
        if (gamertag != value) {
            gamertag = value;
            OnPropertyChanged(nameof(_Gamertag));
        }
    }
}
```

Organización de código

Idioma

El idioma que se utilizará para el desarrollo del código será el inglés.

Llaves

El estilo por emplear para las llaves es el llamado OTBS, es decir, al declarar una clase, método, etc. Las funciones tienen llaves de apertura en la misma línea separadas por un espacio y las llaves que cierran si estarán en su propia línea. Las propiedades que son simples se le puede declarar las llaves de apertura y final sobre la misma línea.

Uso correcto:

```
private void LoginPasswordBox(object sender, KeyEventArgs e) {
    if (e.Key == Key.Enter) {
        SignIn(sender, e);
    }
}
```

Uso no correcto:

```
private void LoginPasswordBox(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
    {
        SignIn(sender, e);
    }
}
```

Comentarios

Se usarán comentarios de una sola línea (//) cuando se trate de explicaciones breves. Cuando sean extensas se usarán los comentarios (/* */). Se colocará un espacio entre el delimitador del

comentario y el texto del comentario. Únicamente se usarán los comentarios cuando sea

necesarios explicar el fundamento del código, puesto que el código se debe de explicar solo. Uso correcto:

```
// Hola mundo

/*    ngerbread  I  love  t  ram  su  jujubes  sweet  roll.  T  ram  su  marz
 *    bear  claw  loll  pop  pastry  dan  sh.  Marshmallow  sesame  snaps
 *    cheesecake  cotton  candy.  I  love  macaroon  muff  n  chocolate  cake
 */
```

Uso no correcto:

```
//Hola mundo

/*Callback */
```

El compilador de C# combina la estructura del código C# con el texto de los comentarios en un único documento XML. La estructura de los comentarios ocupados para la generación de

documentación de API estará definida de la siguiente manera:

Uso correcto:

```
/**
    * <summary>
    * Genera un identificador a las salas que se crean.
    * </summary>
    * <returns>Retorna una cadena que es el identificador de la sala</returns>
    */

/// <summary>
/// Interfaz que contiene los metodos que se encarga de la funcionalidad de la
sala
/// </summary>
```

Uso no correcto:

```
/// <summary> Clase de ejemplo</summary>
///

/*<summary>
 *          * This method is used to generate a unique identifier for the guest
player.
 *                 * </summary>
 *                                  * <returns>Unique identifier for the guest
player.</returns>
 *                                 */
```

Manejo de excepciones

La instrucción "try" debe ser seguida por una llave de apertura en su propia línea. A continuación, el cuerpo de la sentencia deberá ser colocado, seguido de un salto de línea antes de la llave de cierre correspondiente. Posteriormente, se realizará la captura de la excepción, asegurándose de que la sentencia "catch" esté en una línea separada. Es imperativo incluir un bloque "finally" para la destrucción de objetos no necesarios. Es esencial evitar la creación de bloques "try-catch" vacíos.

Se debe evitar descartar excepciones, incluso si se considera improbable que ocurra algún fallo en el sistema. Si es necesario dejar una excepción sin capturar, se debe proporcionar una justificación en un comentario que explique el motivo de esta decisión. Se recomienda evitar el uso de excepciones genéricas (como "Exception") debido a que esto capturaría todas las excepciones, dificultando la identificación precisa del error en situaciones donde solo se requiere manejar una excepción específica.

Ejemplo:

```
public string CreateRoom(string gamertag) {
```

```
    try {
        string idRoom;
        do {
            idRoom = GenerateId();
        } while (_rooms.Contains(idRoom));
        _rooms.Add(idRoom);
        return idRoom;
    } catch (CommunicationException ex) {
        MessageBox.Show($"Error al crear la sala: {ex.Message}");
        return null;
    }
}
```

Prácticas de seguridad

Consultas parametrizas: El uso de consultas parametrizadas ayuda a prevenir la inyección de SQL, que es un vector común de ataque. Al usar consultas parametrizadas, los valores proporcionados por los usuarios se tratan como datos y no como parte de la consulta SQL en sí.

Uso correcto:

```
string query = "INSERT INTO Usuar os (Nombre, Apell do) VALUES (@Nombre, @Apell do)";
    using(SqlConnection connection = new SqlConnection(connectionString)) {
using(SqlCommand command = new SqlCommand(query, connection)) {
    command.flarameters.AddWathValue("@Nombre", nombreUsuario);
command.flarameters.AddWathValue("@Apellido", apellidoUsuario);
    connection.Open(); command.ExecuteNonQuery();
    }
}
```

Uso no correcto:

```
        string query = "INSERT INTO Usuar os (Nombre, Apell do) VALUES ('" + nombreUsuario + "', '" + apellidoUsuario + "')";
        using(SqlConnection connection = new SqlConnection(connectionString)) { using(SqlCommand command = new SqlCommand(query, connection)) {
            connection.Open(); command.ExecuteNonQuery();
        }
```

Encriptación: Se utilizará el cifrado adecuado para proteger datos sensibles, ya sea en reposo (almacenamiento) o en tránsito (comunicaciones). Utilizando algoritmos de encriptación fuertes, la realización de hash seguras como lo es bcrypt.

Ejemplo:

```
public int AddUserAccount(UserModel userModel) {
    try {
        using (var context = new ChineseCheckersEntities())
        using (var transaction = context.Database.BeginTransaction()) {
            var userAccountGame = new UserAccount {
                GamertagUser = userModel.Gamertag,
                Email = userModel.Email,
                Password = BCrypt.Net.BCrypt.HashPassword(userModel.Password),
                ImageProfile = userModel.ImageProfile,
            };
```

```
            context.UserAccount.Add(userAccountGame);
            try {
                context.SaveChanges();
            } catch (DbUpdateException ex) {
                transaction.Rollback();
                MessageBox.Show($"Error de concurrencia al agregar el usuario:
{ex.Message}");
                return -1;
            }
            transaction.Commit();
        }
    } catch (EntityException ex) {
        MessageBox.Show($"Error al agregar la cuenta de usuario: {ex.Message}");
        return -1;
    }
    return 1;
}
```

Referencias

- Meneses, S. (2021, October 5). La Importancia de los Estándares de Código. The Dojo MX Blog. https://blog.thedojo.mx/2021/10/05/estandares-de-calidad-en-el-software.html
- Indentación estilo K&R y variantes. (n.d.). Com.ar. Retrieved August 24, 2023, from http://www.luchonet.com.ar/aed/?page_id=165
- BillWagner. (n.d.). Convenciones de codificación de C# de documentación de .NET. Microsoft.com.
- Retrieved August 24, 2023, from https://learn.microsoft.com/es-es/dotnet/csharp/fundamentals/coding-style/coding-conventions
- BillWagner. (n.d.-b). Instrucciones de control de excepciones: throw y try, catch, finally. Microsoft.com. Retrieved August 24, 2023, from https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/statements/exception-handling-statements