

El primero que utilice fue ChatGPT

El código que me dio fue esto:

```
private string tipoLicencia;
public string TipoLicencia
{
    get { return tipoLicencia; }
    set
    {
        if (!EdadParaLicencia(Edad, value))
        {
            throw new Exception($"La edad no es adecuada para el tipo de licencia
{value}");
        }
        tipoLicencia = value;
    }
}
```

Y esta fue la explicación:

□ **Encapsulamiento:**

- Al usar un **setter** para TipoLicencia, no solo permites que alguien cambie el valor directamente, sino que controlas cómo y cuándo se puede modificar. Esto ayuda a **proteger el estado interno** del objeto.
- El hecho de **validar** la edad contra el tipo de licencia dentro del **setter** asegura que **no se pueda asignar un valor inválido** a la propiedad. Es decir, si alguien intenta asignar una licencia que no es válida para la edad del chofer, el sistema lo detecta en tiempo de ejecución y lanza un error claro (en este caso, una ArgumentException).

□ **Revalidación de datos:**

- Cuando creamos el chofer en el **constructor**, ya validamos la licencia con la edad. Sin embargo, si más tarde **cambiamos la licencia**, necesitamos asegurarnos de que la nueva licencia también sea válida para la edad actual.
- La **validación en el setter** se asegura de que **cualquier cambio posterior** a la propiedad TipoLicencia siga las mismas reglas que aplicamos inicialmente en el constructor. Esto proporciona una **validación continua**.

□ **Centralización de la lógica de negocio:**

- Al colocar la lógica de validación dentro del **setter**, estás centralizando la **lógica de negocio** que determina si un chofer puede tener determinada licencia según su edad.
- Esto tiene la ventaja de que **si la regla cambia** (por ejemplo, si la edad mínima para una licencia "B" cambia), solo necesitas modificar la lógica en un lugar: dentro del método de validación, en lugar de tener múltiples comprobaciones en varias partes del código.

□ **Claridad y mantenimiento del código:**

- Usar un **setter** para la validación mejora la **legibilidad** del código. Cualquier programador que vea tu clase sabrá que la propiedad TipoLicencia tiene **reglas específicas** que se aplican cada vez que se cambia su valor.
- La **mejora en el mantenimiento** también es clave, porque si necesitas hacer cambios en la forma en que validas las licencias, solo modificarías el setter y la función de validación, sin necesidad de alterar el resto del código.

□ **Mejora en la detección de errores:**

- Al lanzar una excepción cuando se intenta asignar un valor inválido a la propiedad TipoLicencia, evitas que el programa entre en un estado inconsistente. El manejo de excepciones permite que el sistema se detenga en el momento en que se produce un error, lo que facilita la depuración.

El segundo que utilice fue Copilot:

Y este fue el código que me dio

// Propiedades

```
public int Edad { get; private set; }
```

```
private string _tipoDeLicencia;
```

// Evento para manejar validaciones

```
public event EventHandler<string> ValidarLicencia;
```

// Propiedad con lógica de validación basada en evento

```
public string TipoDeLicencia
```

```
{
```

```
    get => _tipoDeLicencia;
```

```
    set
```

```
    {
```

```
        if (ValidarLicencia != null)
```

```
        {
```

```
            // Llama al evento y valida
```

```
            ValidarLicencia.Invoke(this, value);
```

```
        }
```

```
        // Si no hay excepción, actualiza la propiedad
```

```
        _tipoDeLicencia = value;
```

```
    }
```

```
}
```

Y esta fue la explicación

□ **Encapsulación de datos:**

- La propiedad Edad utiliza el modificador private set para que solo sea asignada dentro del constructor de la clase, evitando cambios accidentales desde fuera de la clase.
- Esto garantiza que los datos importantes (como la edad, que es clave en la validación) sean consistentes y controlados.

□ **Propiedad privada para manejo interno:**

- `_tipoDeLicencia` es un campo privado que se utiliza para almacenar el valor de `TipoDeLicencia`. Esto asegura que el acceso directo al valor sea controlado únicamente a través de la propiedad pública.

□ **Eventos para validación personalizada:**

- Se define el evento `ValidarLicencia`, que permite delegar la lógica de validación fuera de la propiedad. Esto hace que el código sea más flexible, ya que se puede vincular cualquier método de validación adicional al evento sin modificar directamente la propiedad.
- Los eventos permiten que otros componentes de tu programa puedan reaccionar a cambios en `TipoDeLicencia`. Por ejemplo, podrías notificar a otras partes del sistema si un chofer cambia de licencia.

□ **Validación centralizada en el setter:**

- La propiedad `TipoDeLicencia` incluye lógica para validar el nuevo valor antes de asignarlo, lo cual evita estados inconsistentes en el objeto.
- El uso de `ValidarLicencia.Invoke(this, value)` asegura que toda validación ocurra de manera dinámica y en tiempo real cada vez que se intenta cambiar la licencia.

□ **Extensibilidad:**

- Este diseño con eventos es excelente para escenarios en los que las reglas de validación pueden cambiar o ampliarse. Simplemente necesitas agregar nuevos manejadores al evento `ValidarLicencia` sin tocar el resto de la clase.

El que mejor me sirvió fue el primero, fue más fácil, entendible y cumplió con lo que necesitaba