SISTEMA DE REGISTRO DE VEHÍCULOS

Trabajo de Curso

Asignatura: Ciencia Computacional Avanzada

Integrantes:

Daniel Ayala

Nicolas Perez

Julio Florez

Profesor:

Jose Marcial Tellez Gomez

Universidad Sergio Arboleda

2026

Resumen

El presente trabajo describe de manera detallada el diseño, desarrollo, implementación y análisis de un sistema de registro de vehículos desarrollado en Python. Este sistema permite gestionar información vehicular mediante el uso de una interfaz gráfica, validación de datos mediante expresiones regulares y almacenamiento persistente en formato JSON. El proyecto se enfoca en garantizar la integridad de los datos, la usabilidad del sistema y la modularidad del código, permitiendo su escalabilidad futura.

1. Introducción

La gestión de información es un elemento fundamental en los sistemas computacionales modernos. En el contexto del registro vehicular, es necesario contar con herramientas que permitan almacenar, validar y consultar datos de manera eficiente. Este proyecto surge como una solución académica que implementa un sistema de registro de vehículos utilizando Python, abordando problemáticas comunes como errores en la digitación, inconsistencia de datos y falta de control en la información almacenada.

2. Planteamiento del Problema

En muchos sistemas manuales de registro de información vehicular, se presentan inconsistencias debido a la ausencia de validaciones automáticas. Esto genera problemas como duplicidad de registros, datos incorrectos y dificultad en la consulta de la información. Por lo tanto, se plantea la necesidad de desarrollar un sistema que permita validar y almacenar correctamente la información.

3. Justificación

El desarrollo de este sistema permite aplicar conocimientos de programación orientada a objetos, estructuras de datos y validación mediante expresiones regulares. Además, contribuye a mejorar la organización de la información y reduce errores humanos en el proceso de registro.

4. Objetivos

Objetivo General:
Desarrollar un sistema de registro de vehículos robusto que permita validar, almacenar y

visualizar información.

Objetivos Específicos:
- Diseñar una interfaz gráfica amigable.
- Implementar validaciones de datos.
- Utilizar almacenamiento en JSON.
- Permitir la gestión de registros.

5. Marco Teórico

La programación orientada a objetos (POO) permite estructurar aplicaciones complejas mediante el uso de clases y objetos. Las expresiones regulares son herramientas que permiten validar patrones de texto, siendo útiles para verificar formatos como correos electrónicos o placas. JSON es un formato de intercambio de datos ligero y ampliamente utilizado. La biblioteca Tkinter facilita la creación de interfaces gráficas en Python.

6. Metodología

Se utilizó una metodología de desarrollo incremental, donde el sistema fue construido por módulos. Cada componente fue desarrollado, probado e integrado de manera progresiva, asegurando su correcto funcionamiento.

7. Arquitectura del Sistema

El sistema sigue una arquitectura modular compuesta por diferentes archivos que cumplen funciones específicas. Esta separación permite mantener el código organizado y facilita su mantenimiento. La arquitectura puede considerarse una implementación simplificada del patrón Modelo-Vista-Controlador.

8. Desarrollo del Sistema

El desarrollo del sistema se divide en varios módulos. El archivo principal gestiona la interfaz gráfica y la interacción con el usuario. El módulo de validación se encarga de verificar que los datos cumplan con los formatos establecidos. El módulo de visualización permite mostrar y eliminar registros. Finalmente, el archivo JSON almacena la información.

9. Validaciones del Sistema

Cada campo del formulario es validado mediante reglas específicas. Estas validaciones aseguran la integridad de los datos y evitan errores en el almacenamiento. Por ejemplo, la placa debe cumplir con un formato específico, el correo debe ser válido y el número de teléfono debe tener una longitud determinada.

Imagen de código respectivo aquí

```python
import re

class Validacion:
    """Clase independiente para validaciones mediante expresiones regulares"""

    @staticmethod
    def validar_placa(placa):
        """
        Valida placa colombiana (formato: ABC123 o ABC12D)
        """
        patron = r'^[A-Z]{3}\d{3}$|^[A-Z]{3}\d{2}[A-Z]$'
        return bool(re.match(patron, placa.upper()))

    @staticmethod
    def validar_marca(marca):
        """
        Valida marca: solo letras y espacios
        """
        patron = r'^[A-Za-záéíóúÁÉÍÓÚñÑ\s]+$'
        return bool(re.match(patron, marca))

    @staticmethod
    def validar_modelo(modelo):
        """
        Valida año: 1900-2026
        """
        try:
            año = int(modelo)
            return 1900 <= año <= 2026
        except ValueError:
            return False

    @staticmethod
    def validar_color(color):
        """
        Valida color: solo letras
        """
        patron = r'^[A-Za-záéíóúÁÉÍÓÚñÑ]+$'
        return bool(re.match(patron, color))

    @staticmethod
    def validar_chasis(chasis):
        """
        Valida número de chasis: alfanumérico de 17 caracteres
```

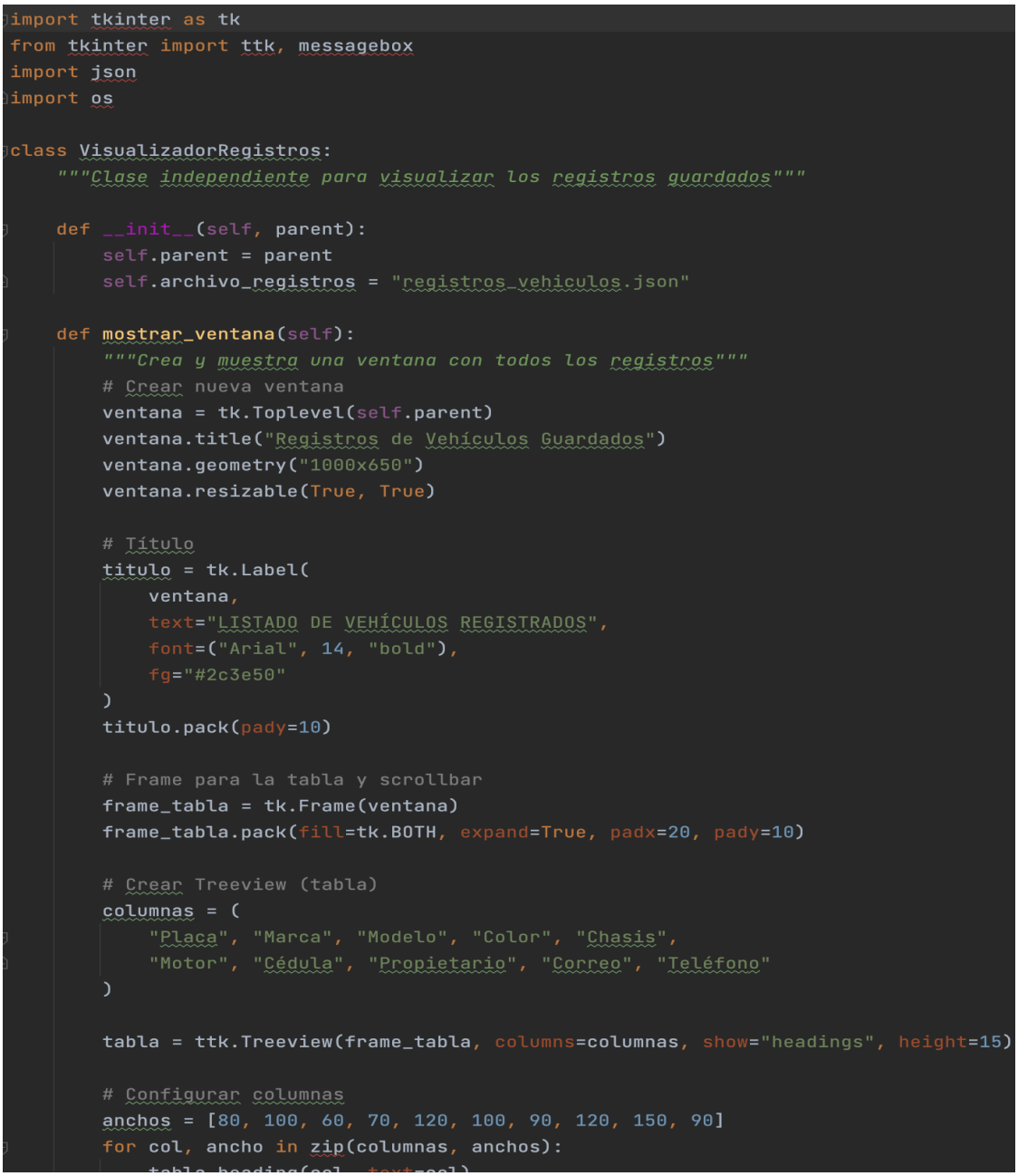
```python
        patron = r'^[A-Za-z0-9]{17}$'
        return bool(re.match(patron, chasis))

    @staticmethod
    def validar_motor(motor):
        """
        Valida número de motor: alfanumérico
        """
        patron = r'^[A-Za-z0-9]+$'
        return bool(re.match(patron, motor))

    @staticmethod
    def validar_cedula(cedula):
        """
        Valida cédula: solo números de 7-10 dígitos
        """
        patron = r'^\d{7,10}$'
        return bool(re.match(patron, cedula))

    @staticmethod
    def validar_nombre(nombre):
        """
        Valida nombre: solo letras y espacios
        """
        patron = r'^[A-Za-záéíóúÁÉÍÓÚñÑ\s]+$'
        return bool(re.match(patron, nombre))

    @staticmethod
    def validar_correo(correo):
        """
        Valida correo electrónico: formato usuario@dominio.com
        """
        patron = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
        return bool(re.match(patron, correo))

    @staticmethod
    def validar_telefono(telefono):
        """
        Valida teléfono: 10 dígitos
        """
        patron = r'^\d{10}$'
        return bool(re.match(patron, telefono))
```

10.

Persistencia de Datos

El sistema utiliza archivos JSON para almacenar la información. Este formato permite guardar datos estructurados de manera sencilla. Cada registro es almacenado como un diccionario dentro de una lista.

Imagen de código respectivo aquí



```python
import tkinter as tk
from tkinter import ttk, messagebox
import json
import os

class VisualizadorRegistros:
    """Clase independiente para visualizar los registros guardados"""

    def __init__(self, parent):
        self.parent = parent
        self.archivo_registros = "registros_vehiculos.json"

    def mostrar_ventana(self):
        """Crea y muestra una ventana con todos los registros"""
        # Crear nueva ventana
        ventana = tk.Toplevel(self.parent)
        ventana.title("Registros de Vehículos Guardados")
        ventana.geometry("1000x650")
        ventana.resizable(True, True)

        # Título
        titulo = tk.Label(
            ventana,
            text="LISTADO DE VEHÍCULOS REGISTRADOS",
            font=("Arial", 14, "bold"),
            fg="#2c3e50"
        )
        titulo.pack(pady=10)

        # Frame para la tabla y scrollbar
        frame_tabla = tk.Frame(ventana)
        frame_tabla.pack(fill=tk.BOTH, expand=True, padx=20, pady=10)

        # Crear Treeview (tabla)
        columnas = (
            "Placa", "Marca", "Modelo", "Color", "Chasis",
            "Motor", "Cédula", "Propietario", "Correo", "Teléfono"
        )

        tabla = ttk.Treeview(frame_tabla, columns=columnas, show="headings", height=15)

        # Configurar columnas
        anchos = [80, 100, 60, 70, 120, 100, 90, 120, 150, 90]
        for col, ancho in zip(columnas, anchos):
            tabla.heading(col, text=col)
```

```
        tabla.heading(col, text=col)
        tabla.column(col, width=ancho, minwidth=ancho)

    # Scrollbars
    scroll_y = ttk.Scrollbar(frame_tabla, orient=tk.VERTICAL, command=tabla.yview)
    scroll_x = ttk.Scrollbar(frame_tabla, orient=tk.HORIZONTAL, command=tabla.xview)
    tabla.configure(yscrollcommand=scroll_y.set, xscrollcommand=scroll_x.set)

    # Posicionar tabla y scrollbars
    tabla.grid(row=0, column=0, sticky="nsew")
    scroll_y.grid(row=0, column=1, sticky="ns")
    scroll_x.grid(row=1, column=0, sticky="ew")

    frame_tabla.grid_rowconfigure(0, weight=1)
    frame_tabla.grid_columnconfigure(0, weight=1)

    # Cargar y mostrar registros
    self.cargar_registros_en_tabla(tabla)

    # Frame para botones
    frame_botones = tk.Frame(ventana)
    frame_botones.pack(pady=10)

    # Botón Actualizar
    btn_actualizar = tk.Button(
        frame_botones,
        text="Actualizar Lista",
        command=lambda: self.actualizar_tabla(tabla, ventana),
        bg="#3498db",
        fg="white",
        font=("Arial", 10),
        width=15
    )
    btn_actualizar.pack(side=tk.LEFT, padx=5)

    # Botón Eliminar Registro Seleccionado
    btn_eliminar = tk.Button(
        frame_botones,
        text="Eliminar Registro",
        command=lambda: self.eliminar_registro_seleccionado(tabla, ventana),
        bg="#e74c3c",
        fg="white",
        font=("Arial", 10),
        width=15
    )
```

```
    btn_eliminar.pack(side=tk.LEFT, padx=5)

    # Botón Cerrar
    btn_cerrar = tk.Button(
        frame_botones,
        text="Cerrar",
        command=ventana.destroy,
        bg="#95a5a6",
        fg="white",
        font=("Arial", 10),
        width=15
    )
    btn_cerrar.pack(side=tk.LEFT, padx=5)

    # Etiqueta con conteo de registros
    self.actualizar_conteo(ventana, tabla)

    # Instrucción de selección
    label_instruccion = tk.Label(
        ventana,
        text="💡 Para eliminar: selecciona un registro haciendo clic en él y presiona 'Eliminar Registro'",
        font=("Arial", 9, "italic"),
        fg="#7f8c8d"
    )
    label_instruccion.pack(pady=2)

def cargar_registros_en_tabla(self, tabla):
    """
    Carga los registros desde el archivo JSON a la tabla
    """
    try:
        # Limpiar tabla
        for item in tabla.get_children():
            tabla.delete(item)

        # Cargar registros
        registros = self.cargar_registros()

        if not registros:
            # Mostrar mensaje si no hay registros
            tabla.insert("", tk.END, values=("No hay registros", "", "", "", "", "", "", "", "", ""))
            return

        # Insertar registros en la tabla
        for registro in registros:
```

```
        for registro in registros:
            # Asegurar que todos los campos existen y convertir a string
            valores = (
                str(registro.get('placa', '')).strip(),
                str(registro.get('marca', '')).strip(),
                str(registro.get('modelo', '')).strip(),
                str(registro.get('color', '')).strip(),
                str(registro.get('chasis', '')).strip(),
                str(registro.get('motor', '')).strip(),
                str(registro.get('cedula', '')).strip(),
                str(registro.get('nombre', '')).strip(),
                str(registro.get('correo', '')).strip(),
                str(registro.get('telefono', '')).strip()
            )
            tabla.insert("", tk.END, values=valores)

        print(f"✅ {len(registros)} registros cargados correctamente")

    except Exception as e:
        print(f"❌ Error al cargar registros: {e}")
        messagebox.showerror("Error", f"Error al cargar registros: {str(e)}")

def cargar_registros(self):
    """
    Carga los registros desde el archivo JSON
    """
    if os.path.exists(self.archivo_registros):
        try:
            with open(self.archivo_registros, 'r', encoding='utf-8') as file:
                contenido = file.read().strip()
                if contenido:
                    return json.loads(contenido)
                else:
                    return []
        except json.JSONDecodeError as e:
            print(f"❌ Error al decodificar JSON: {e}")
            return []
        except Exception as e:
            print(f"❌ Error al leer archivo: {e}")
            return []
    else:
        print(f"📁 Archivo {self.archivo_registros} no existe. Creando nuevo...")
        return []

def guardar_registro(self, datos_vehiculo):
```

```
    Guarda un nuevo registro en el archivo JSON
    """
    try:
        registros = self.cargar_registros()
        registros.append(datos_vehiculo)

        with open(self.archivo_registros, 'w', encoding='utf-8') as file:
            json.dump(registros, file, ensure_ascii=False, indent=4)

        print(f"✅ Registro guardado: {datos_vehiculo.get('placa', '')}")
        return True

    except Exception as e:
        print(f"❌ Error al guardar registro: {e}")
        return False

def eliminar_registro_seleccionado(self, tabla, ventana):
    """
    Elimina el registro seleccionado de la tabla y del archivo
    """
    # Obtener el elemento seleccionado
    seleccion = tabla.selection()

    if not seleccion:
        messagebox.showwarning(
            "Sin selección",
            "Por favor, selecciona un registro para eliminar."
        )
        return

    # Obtener valores del registro seleccionado
    item = tabla.item(seleccion[0])
    valores = item['values']

    # Verificar si es el mensaje de "No hay registros"
    if valores[0] == "No hay registros":
        messagebox.showinfo("Información", "No hay registros para eliminar.")
        return

    # Crear un diccionario con los datos del registro seleccionado
    registro_seleccionado = {
        'placa': str(valores[0]).strip(),
        'marca': str(valores[1]).strip(),
        'modelo': str(valores[2]).strip(),
        'color': str(valores[3]).strip(),
```

```
        'chasis': str(valores[4]).strip(),
        'motor': str(valores[5]).strip(),
        'cedula': str(valores[6]).strip(),
        'nombre': str(valores[7]).strip(),
        'correo': str(valores[8]).strip(),
        'telefono': str(valores[9]).strip()
    }

    # Mostrar información del registro a eliminar
    mensaje_confirmacion = f"¿Estás seguro de eliminar el siguiente registro?\n\n"
    mensaje_confirmacion += f"🔲 Placa: {valores[0]}\n"
    mensaje_confirmacion += f"👤 Propietario: {valores[7]}\n"
    mensaje_confirmacion += f"🚗 Marca: {valores[1]} - Modelo: {valores[2]}\n"
    mensaje_confirmacion += f"🎨 Color: {valores[3]}"

    # Confirmar eliminación
    respuesta = messagebox.askyesno(
        "Confirmar eliminación",
        mensaje_confirmacion,
        icon='warning'
    )

    if respuesta:
        # Cargar todos los registros
        registros = self.cargar_registros()

        if not registros:
            messagebox.showerror("Error", "No hay registros para eliminar.")
            return

        # Buscar y eliminar el registro correspondiente
        registros_actualizados = []
        eliminado = False
        registro_encontrado = None

        for registro in registros:
            # Comparar todos los campos importantes
            placa_coincide = str(registro.get('placa', '')).strip() == registro_seleccionado['placa']
            cedula_coincide = str(registro.get('cedula', '')).strip() == registro_seleccionado['cedula']
            chasis_coincide = str(registro.get('chasis', '')).strip() == registro_seleccionado['chasis']

            # Si coinciden placa Y (cédula O chasis), consideramos que es el mismo registro
            if placa_coincide and (cedula_coincide or chasis_coincide):
                eliminado = True
                registro_encontrado = registro
```

```
                eliminado = True
                registro_encontrado = registro
                print(f"🗑️ Registro eliminado: {registro}")
                continue

            registros_actualizados.append(registro)

        if eliminado:
            # Guardar registros actualizados
            try:
                with open(self.archivo_registros, 'w', encoding='utf-8') as file:
                    json.dump(registros_actualizados, file, ensure_ascii=False, indent=4)

                # Actualizar tabla
                self.actualizar_tabla(tabla, ventana)

                messagebox.showinfo(
                    "✅ Registro eliminado",
                    "El registro ha sido eliminado correctamente."
                )

            except Exception as e:
                messagebox.showerror(
                    "❌ Error",
                    f"No se pudo eliminar el registro: {str(e)}"
                )
        else:
            # Mostrar información de depuración
            print("🔍 Registro buscado:", registro_seleccionado)
            print("📋 Registros en archivo:", registros)

            messagebox.showerror(
                "❌ Error",
                "No se pudo encontrar el registro para eliminar.\n"
                "Por favor, actualiza la lista e intenta nuevamente."
            )

def actualizar_tabla(self, tabla, ventana):
    """
    Actualiza la tabla con los registros más recientes
    """
    self.cargar_registros_en_tabla(tabla)
    self.actualizar_conteo(ventana, tabla)
```

```python
def actualizar_tabla(self, tabla, ventana):
    """
    Actualiza la tabla con los registros más recientes
    """
    self.cargar_registros_en_tabla(tabla)
    self.actualizar_conteo(ventana, tabla)

def actualizar_conteo(self, ventana, tabla):
    """
    Actualiza el contador de registros
    """
    total_registros = len(tabla.get_children())

    # Verificar si hay registros reales (no el mensaje de "No hay registros")
    if total_registros == 1:
        item = tabla.get_children()[0]
        valores = tabla.item(item)['values']
        if valores[0] == "No hay registros":
            total_registros = 0

    # Eliminar label anterior si existe
    for widget in ventana.winfo_children():
        if isinstance(widget, tk.Label) and "registros encontrados" in widget.cget("text"):
            widget.destroy()

    # Crear nuevo label con conteo
    label_conteo = tk.Label(
        ventana,
        text=f"Total de registros: {total_registros} vehículo(s) encontrado(s)",
        font=("Arial", 10, "italic"),
        fg="#7f8c8d"
    )
    label_conteo.pack(pady=5)
```

11. Interfaz Gráfica

La interfaz gráfica fue desarrollada utilizando Tkinter. Se diseñaron formularios para la captura de datos y tablas para la visualización de registros. Esto permite una interacción intuitiva con el usuario.

Imagen de código respectivo aquí

```python
import tkinter as tk
from tkinter import messagebox
from validacion import Validacion
from visualizador_registros import VisualizadorRegistros

class RegistroVehiculoApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Registro de Vehículo")
        self.root.geometry("500x750")
        self.root.resizable(False, False)

        # Diccionario para almacenar los campos de entrada
        self.entries = {}

        # Inicializar visualizador de registros
        self.visualizador = VisualizadorRegistros(root)  # NUEVA LÍNEA

        # Crear los campos del formulario
        self.crear_formulario()

        # Botones
        self.crear_botones()

    def crear_formulario(self):
        """Crea todos los campos del formulario"""
        campos = [
            ("Placa del vehículo:", "placa"),
            ("Marca:", "marca"),
            ("Modelo (año):", "modelo"),
            ("Color:", "color"),
            ("Número de chasis:", "chasis"),
            ("Número de motor:", "motor"),
            ("Cédula del propietario:", "cedula"),
            ("Nombre del propietario:", "nombre"),
            ("Correo electrónico:", "correo"),
            ("Teléfono de contacto:", "telefono")
        ]

        for i, (label_text, campo) in enumerate(campos):
            # Label
            label = tk.Label(self.root, text=label_text, font=("Arial", 10))
            label.place(x=50, y=30 + i*60)

            # Entry
```

```
            # Entry
            entry = tk.Entry(self.root, font=("Arial", 10), width=30)
            entry.place(x=250, y=30 + i*60)

            # Label para mensaje de error
            error_label = tk.Label(self.root, text="", fg="red", font=("Arial", 8))
            error_label.place(x=250, y=55 + i*60)

            # Guardar referencia
            self.entries[campo] = {
                'entry': entry,
                'error_label': error_label
            }

    def crear_botones(self):
        """Crea los botones del formulario"""
        # Botón Guardar
        btn_guardar = tk.Button(
            self.root,
            text="Guardar Registro",
            command=self.validar_registro,
            bg="#4CAF50",
            fg="white",
            font=("Arial", 11),
            width=15
        )
        btn_guardar.place(x=120, y=660)

        # Botón Limpiar
        btn_limpiar = tk.Button(
            self.root,
            text="Limpiar",
            command=self.limpiar_campos,
            bg="#f44336",
            fg="white",
            font=("Arial", 11),
            width=15
        )
        btn_limpiar.place(x=250, y=660)

        # Botón Ver Registros
        btn_ver_registros = tk.Button(
            self.root,
            text="Ver Registros Guardados",
            command=self.visualizador.mostrar_ventana,
```

```
            bg="#3498db",
            fg="white",
            font=("Arial", 11),
            width=32
        )
        btn_ver_registros.place(x=120, y=700)

    def limpiar_campos(self):
        """Limpia todos los campos del formulario y mensajes de error"""
        for campo in self.entries.values():
            campo['entry'].delete(0, tk.END)
            campo['entry'].config(bg="white")
            campo['error_label'].config(text="")

    def validar_registro(self):
        """Valida todos los campos del formulario"""
        resultados_validacion = {}
        campos_validos = True
        datos_vehiculo = {}  # diccionario para guardar datos


        # Validar cada campo
        for campo, datos in self.entries.items():
            valor = datos['entry'].get().strip()

            # Limpiar mensajes de error anteriores
            datos['error_label'].config(text="")
            datos['entry'].config(bg="white")

            # Verificar campos vacíos
            if not valor:
                self.mostrar_error(campo, "Campo obligatorio")
                valido = False
            else:
                # Validar según el campo
                if campo == 'placa':
                    valido = Validacion.validar_placa(valor)
                    if not valido:
                        self.mostrar_error(campo, "Formato inválido. Ej: ABC123 o ABC12D")

                elif campo == 'marca':
                    valido = Validacion.validar_marca(valor)
                    if not valido:
                        self.mostrar_error(campo, "Solo letras y espacios")
```

```
        elif campo == 'modelo':
            valido = Validacion.validar_modelo(valor)
            if not valido:
                self.mostrar_error(campo, "Año entre 1900 y 2026")

        elif campo == 'color':
            valido = Validacion.validar_color(valor)
            if not valido:
                self.mostrar_error(campo, "Solo letras")

        elif campo == 'chasis':
            valido = Validacion.validar_chasis(valor)
            if not valido:
                self.mostrar_error(campo, "17 caracteres alfanuméricos")

        elif campo == 'motor':
            valido = Validacion.validar_motor(valor)
            if not valido:
                self.mostrar_error(campo, "Solo caracteres alfanuméricos")

        elif campo == 'cedula':
            valido = Validacion.validar_cedula(valor)
            if not valido:
                self.mostrar_error(campo, "7-10 dígitos numéricos")

        elif campo == 'nombre':
            valido = Validacion.validar_nombre(valor)
            if not valido:
                self.mostrar_error(campo, "Solo letras y espacios")

        elif campo == 'correo':
            valido = Validacion.validar_correo(valor)
            if not valido:
                self.mostrar_error(campo, "Formato: usuario@dominio.com")

        elif campo == 'telefono':
            valido = Validacion.validar_telefono(valor)
            if not valido:
                self.mostrar_error(campo, "10 dígitos numéricos")

    # Guardar datos del vehículo (solo si el campo tiene valor)
    if valor:
        datos_vehiculo[campo] = valor

    resultados_validacion[campo] = valido
```

```
            if not valido:
                campos_validos = False

        # Mostrar resultado final y guardar si es válido
        if campos_validos:
            # Guardar el registro antes de mostrar éxito
            if self.visualizador.guardar_registro(datos_vehiculo):
                self.mostrar_exito()
            else:
                messagebox.showerror("Error", "No se pudo guardar el registro")

    def mostrar_error(self, campo, mensaje):
        """Muestra error en el campo específico"""
        if campo in self.entries:
            self.entries[campo]['error_label'].config(text=mensaje)
            self.entries[campo]['entry'].config(bg="#FFE4E1")

    def mostrar_exito(self):
        """Muestra mensaje de registro exitoso"""
        messagebox.showinfo(
            "Registro Exitoso",
            "¡El vehículo ha sido registrado correctamente!"
        )
        self.limpiar_campos()

def main():
    """Función principal"""
    root = tk.Tk()
    app = RegistroVehiculoApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

12. Resultados

El sistema permite registrar información de manera eficiente, validando los datos y almacenándolos correctamente. Además, facilita la visualización y eliminación de registros.

13. Análisis

El sistema presenta un buen nivel de modularidad y funcionalidad. Sin embargo, existen áreas de mejora como la implementación de una base de datos, la prevención de duplicados y el uso de identificadores únicos.

14. Pruebas del Sistema

Se realizaron pruebas funcionales para verificar el correcto funcionamiento del sistema. Se probaron diferentes casos de entrada, incluyendo datos válidos e inválidos, para asegurar que las validaciones funcionaran correctamente.

15. Limitaciones

El sistema presenta limitaciones como el uso de archivos JSON en lugar de bases de datos, lo cual puede afectar la escalabilidad. Además, no se implementa control de concurrencia.

16. Conclusiones

El proyecto cumple con los objetivos planteados, demostrando la aplicación de conceptos de programación avanzada. Se destaca la importancia de la validación de datos y la modularidad del código.

17. Recomendaciones

Se recomienda implementar mejoras como el uso de bases de datos, validación de duplicados, mejoras en la interfaz y nuevas funcionalidades.

18. Referencias

American Psychological Association. (2020). Publication manual of the American Psychological Association (7th ed.).