Documentación Técnica

Sistema de Gestión 'RentaYa'

Trabajo Práctico Integrador - DAO

Grupo 29

Integrantes:

• Castro Maximiliano

• Pereira Puca Nicolas Francisco

• Koncurat Joaquín Ernesto

Alcance 1: Administrar Entidades Principales (ABM)

Requisito Propuesto: "Altas, bajas y modificaciones de clientes, vehículos y empleados, con validaciones de unicidad."

Cumplimiento Técnico: El sistema implementa la gestión de estas entidades a través de la capa de Servicios, que actúa como "guardián" de las reglas de negocio antes de interactuar con la base de datos. Se utiliza Baja Lógica (soft delete) para mantener la integridad referencial del historial.

1.1. Gestión de Clientes

La lógica de clientes asegura que no existan duplicados en el sistema basándose en el DNI, que actúa como identificador único natural.

Validación de Unicidad: Antes de crear un cliente, el servicio consulta al repositorio si el DNI ya existe. Además, captura errores de integridad de la base de datos.

Validación de Datos: Se verifica formato de email y completitud de campos.

Fragmento de Código (src/services/cliente_service.py):

```
def crear_cliente(self, nombre, dni, email, telefono):
    # ... validaciones previas ...

    try:
        # Intenta insertar en la base de datos
        return self._repo.crear(cliente)

    except IntegrityError:
        # Captura la restricción UNIQUE definida en la tabla SQL
        raise Exception("Ya existe un cliente registrado con ese DNI.")
```

1.2. Gestión de Vehículos (Con Patrón Factory)

Para los vehículos, la validación principal es la Patente. Además, aquí se aplica el patrón de diseño **Factory Method** para instanciar la clase correcta.

Validación de Unicidad: Se busca explícitamente la patente antes de intentar la creación.

Creación Desacoplada: El servicio no hace *new Vehiculo()*, sino que delega esa tarea a *VehiculoFactory*.

Fragmento de Código (src/services/vehiculo_service.py):

```
@staticmethod
def crear_vehiculo(usuario_actual, patente, ...):
    # 1. Validación de formato (Regex) y unicidad
    if not VehiculoService._patente_valida(patente):
        return False, "Formato de patente inválido."

    existente = VehiculoRepository.buscar_por_patente(patente)
    if existente is not None:
        return False, "Ya existe un vehículo con esa patente." # [Cumple validación de
unicidad]
```

```
    # 2. Creación del objeto usando Factory Method
    try:
        vehiculo = VehiculoFactory.get_vehiculo(tipo=tipo, patente=patente, ...)
    except ValueError as e:
        return False, str(e)

    # 3. Persistencia
    VehiculoRepository.crear(vehiculo)
    return True, vehiculo
```

1.3. Gestión de Empleados y Seguridad

Los empleados requieren validaciones más estrictas, ya que tienen acceso al sistema. Se valida la unicidad tanto del DNI como del Nombre de Usuario.

Validación: Se impide el registro si el DNI ya está en uso por otro empleado.

Roles: Se asigna obligatoriamente un rol (ADMIN o EMPLEADO) que determinará los permisos en el Dashboard.

Fragmento de Código (src/services/empleado_service.py):

```
@staticmethod
def crear_empleado(nombre, apellido, dni, usuario, ...):
    # Validación manual antes de ir a la BD
    existente = EmpleadoRepository.buscar_por_dni(dni)
    if existente is not None:
        return False, "Ya existe un empleado registrado con ese DNI."

    # ... creación del objeto Empleado ...

    try:
        EmpleadoRepository.crear(empleado)
        return True, empleado
    except sqlite3.IntegrityError as e:
        # Captura si el USUARIO (unique) ya existe
        if "usuario" in str(e).lower():
            return False, "El nombre de usuario ya está en uso."
```

1.4. Estrategia de Baja Lógica (Soft Delete)

Para cumplir con el requisito de "Integridad de datos", el sistema NO elimina registros físicos de la base de datos (DELETE). En su lugar, utiliza una columna activo (booleano/int).

Esto permite que, si un cliente o vehículo tiene alquileres pasados, los reportes históricos sigan funcionando correctamente aunque la entidad ya no esté disponible para nuevas operaciones.

Fragmento de Código (Ejemplo en VehiculoRepository):

```
@staticmethod
def inactivar(id_vehiculo):
    conn = get_connection()
```

```
    cursor = conn.cursor()
    # Solo actualizamos el estado, no borramos la fila
    cursor.execute("UPDATE vehiculos SET activo = 0 WHERE id_vehiculo = ?", (id_vehiculo,))
    conn.commit()
```

Validación en Listados: Todos los métodos listar() filtran automáticamente los inactivos para las operaciones diarias:

```
cursor.execute("SELECT * FROM vehiculos WHERE activo = 1")
```

Alcance 2: Registrar y Controlar Alquileres

Requisito Propuesto: "Gestión completa del proceso de alquiler (inicio, devolución, cálculo de costos, control de kilometraje y combustible). Validación de disponibilidad del vehículo según fechas y mantenimiento."

Cumplimiento Técnico: El ciclo de vida del alquiler se divide en dos fases críticas: la Apertura, donde se bloquea el recurso para evitar conflictos, y el Cierre, donde se procesa la facturación y se actualiza el estado de la flota.

2.1. Validación de Disponibilidad (Apertura)

Para garantizar que un vehículo no se alquile dos veces en la misma fecha ni se entregue si está en taller, el servicio realiza tres verificaciones secuenciales antes de confirmar la operación.

• **Validación de Fechas:** Se asegura la coherencia temporal (la fecha de fin no puede ser anterior a la de inicio).

• **Validación de Estado:** Se verifica que el vehículo no esté marcado como "MANTENIMIENTO".

• **Validación de Solapamiento:** Se consulta a la base de datos si existe algún alquiler activo ('ABIERTO') cuyo rango de fechas choque con el solicitado.

Fragmento de Código (src/services/alquiler_service.py):

```
@staticmethod
def crear_alquiler(id_cliente, id_vehiculo, ...):
    # 1. Validar coherencia de fechas
    if fecha_fin < fecha_inicio:
        return False, "La fecha de fin no puede ser anterior a la de inicio."

    # 2. Validar estado actual del vehículo
    vehiculo = VehiculoRepository.obtener_por_id(id_vehiculo)
    if vehiculo.estado == "MANTENIMIENTO":
        return False, "El vehículo está en mantenimiento y no puede ser alquilado."

    # 3. Validar solapamiento de fechas (Disponibilidad real)
    if AlquilerRepository.existe_alquiler_activo_en_rango(
        id_vehiculo, fecha_inicio_iso, fecha_fin_iso
    ):
        return False, "El vehículo ya tiene un alquiler activo en ese rango de fechas."

    # Si pasa todas las validaciones, se crea el alquiler
    alquiler_creado = AlquilerRepository.crear(alquiler)

    # Y se actualiza el estado del vehículo para bloquearlo visualmente
    vehiculo.estado = "ALQUILADO"
    VehiculoRepository.actualizar(vehiculo)
```

2.2. Gestión de Cierre y Facturación

El cierre del alquiler no es solo marcar un registro como "terminado". El sistema realiza un cálculo automático para asegurar la integridad financiera y operativa.

Cálculo de Costos: *(Días Reales × Precio por Día) + Monto Extra*. Esto permite cobrar el tiempo exacto de uso más multas o recargos si los hubiera.

Actualización de Flota: Se actualiza el kilometraje y combustible del vehículo con los valores de retorno, lo cual es vital para el control de mantenimiento futuro.

Liberación: El vehículo pasa automáticamente a estado "DISPONIBLE".

Fragmento de Código (src/services/alquiler_service.py):

```
@staticmethod
def cerrar_alquiler(id_alquiler, ...):
    # ... validaciones previas ...

    # 1. Calcular costo base por los días utilizados
    dias_reales = (fecha_devolucion - fecha_inicio).days
    costo_base = dias_reales * vehiculo.precio_por_dia

    # 2. Calcular total final sumando extras (multas, recargos)
    total = costo_base + monto_extra

    # 3. Validación de consistencia de kilometraje
    if km_final < alquiler.km_inicial:
        return False, "El KM final no puede ser menor al inicial."

    # 4. Actualizar Alquiler (Cierre financiero)
    alquiler.total = total
    alquiler.estado = "CERRADO"
    AlquilerRepository.actualizar_cierre(alquiler)

    # 5. Actualizar Vehículo (Liberación operativa)
    vehiculo.km_actual = km_final
    vehiculo.combustible_actual = combustible_final
    vehiculo.estado = "DISPONIBLE"
    VehiculoRepository.actualizar(vehiculo)
```

2.3. Consulta de Disponibilidad (SQL)

La lógica de "no solapamiento" es compleja y se maneja eficientemente en la capa de datos para evitar traer todos los registros a memoria.

Fragmento de Código (src/repositories/alquiler_repository.py):

```
@staticmethod
def existe_alquiler_activo_en_rango(id_vehiculo, fecha_desde, fecha_hasta):
    # Lógica de solapamiento: NO (Fin < NuevoInicio O Inicio > NuevoFin)
    cursor.execute(
        """
        SELECT COUNT(*)
        FROM alquileres
        WHERE id_vehiculo = ?
         AND estado = 'ABIERTO'
         AND NOT (fecha_fin < ? OR fecha_inicio > ?)
        """,
        (id_vehiculo, fecha_desde, fecha_hasta),
    )
```

```
    return cursor.fetchone()[0] > 0
```

Alcance 3: Registrar Mantenimientos y Bloqueos

Requisito Propuesto: "Carga de mantenimientos preventivos y correctivos, evitando su alquiler durante períodos de reparación o revisión."

Cumplimiento Técnico: El sistema trata al "Mantenimiento" como un evento de bloqueo en el calendario del vehículo. A diferencia de un alquiler, no genera ingresos, pero ocupa la disponibilidad del recurso.

3.1. Lógica de Bloqueo Preventivo

Para registrar un mantenimiento, el sistema debe garantizar que el vehículo esté físicamente disponible. No se puede enviar al taller un auto que está actualmente alquilado por un cliente.

El servicio de mantenimiento realiza una **Doble Validación de Solapamiento**:

• **Contra otros Mantenimientos:** Evita duplicar órdenes de trabajo en las mismas fechas.

• **Contra Alquileres Activos:** Verifica que no exista un contrato de alquiler 'ABIERTO' que coincida con las fechas de reparación.

Fragmento de Código (src/services/mantenimiento_service.py):

```
@staticmethod
def crear_mantenimiento(id_vehiculo, fecha_inicio_str, fecha_fin_str, descripcion):
    # ... validaciones de formato de fecha ...

    # 1. Validar solapamiento con otros MANTENIMIENTOS existentes
    mantenimientos = MantenimientoRepository.listar_por_vehiculo(id_vehiculo)
    for m in mantenimientos:
        # Parsea fechas y verifica si el rango choca
        if MantenimientoService._rango_se_solapa(fecha_inicio, fecha_fin, m_ini, m_fin):
            return False, "Ya existe un mantenimiento en ese rango."

    # 2. Validar solapamiento con ALQUILERES activos
    alquileres = AlquilerRepository.listar_por_vehiculo(id_vehiculo)
    for a in alquileres:
        if a.estado != "ABIERTO": continue # Solo importan los activos

        # Verifica si el auto está alquilado en esas fechas
        if MantenimientoService._rango_se_solapa(fecha_inicio, fecha_fin, a_ini, a_fin):
            return False, "El vehículo tiene alquileres activos en ese rango."

    # Si está libre, se crea el bloqueo
    mantenimiento = Mantenimiento(..., id_vehiculo=id_vehiculo, ...)
    MantenimientoRepository.crear(mantenimiento)

    return True, mantenimiento
```

3.2. Algoritmo de Comparación de Fechas

Para determinar si dos rangos de fechas chocan, se implementó una función utilitaria estática que simplifica la lógica temporal.

Lógica: Dos rangos se solapan si **NO** ocurre que uno termine antes de que empiece el otro, o que uno empiece después de que el otro termine.

Fragmento de Código (src/services/mantenimiento_service.py):

```
@staticmethod
def _rango_se_solapa(desde1, hasta1, desde2, hasta2):
    # Retorna True si el rango [desde1, hasta1] se superpone con [desde2, hasta2].
    # Lógica inversa: Si (Termina1 < Empieza2) O (Empieza1 > Termina2) -> No se tocan.
    # Negamos eso para saber si SE tocan.
    return not (hasta1 < desde2 or hasta2 < desde1)
```

3.3. Persistencia y Relaciones

Los mantenimientos se guardan en una tabla dedicada que vincula el vehículo mediante clave foránea, permitiendo generar historiales de reparaciones por unidad.

Estructura de Datos (src/repositories/db_connection.py - Esquema):

```
CREATE TABLE IF NOT EXISTS mantenimientos (
    id_mantenimiento INTEGER PRIMARY KEY AUTOINCREMENT,
    id_vehiculo     INTEGER NOT NULL,
    fecha_inicio     TEXT    NOT NULL, -- Formato ISO 'YYYY-MM-DD'
    fecha_fin        TEXT    NOT NULL,
    descripcion     TEXT    NOT NULL,

    FOREIGN KEY (id_vehiculo) REFERENCES vehiculos(id_vehiculo)
);
```

Alcance 4: Controlar Incidentes

Requisito Propuesto: "Registro de multas y daños asociados a un alquiler, con seguimiento de su estado de cobro."

Cumplimiento Técnico: El sistema modela los incidentes como entidades dependientes de un alquiler específico. Esto permite una trazabilidad completa: se sabe no solo qué vehículo sufrió el daño, sino qué cliente lo tenía en ese momento y durante qué contrato.

Se implementó un Ciclo de Vida de Estado (PENDIENTE -> PAGADO) para distinguir entre la deuda registrada y el dinero efectivamente ingresado a la caja.

4.1. Registro y Asociación

Al crear un incidente, el sistema lo vincula obligatoriamente a un alquiler existente mediante su ID (id_alquiler). Esto actúa como una Clave Foránea lógica que conecta el incidente con el cliente y el vehículo involucrados.

Se validan los datos monetarios para evitar inconsistencias (montos negativos) y se categoriza el incidente (MULTA, DAÑO, OTRO) para futuros reportes estadísticos.

Fragmento de Código (src/services/incidente_service.py):

```
@staticmethod
def crear_incidente(id_alquiler, tipo, descripcion, monto):
    # 1. Validar existencia del alquiler padre
    ok, alquiler = IncidenteService._obtener_alquiler_valido(id_alquiler)
    if not ok:
        return False, "El alquiler indicado no existe."

    # 2. Validar coherencia del monto
    if monto < 0:
        return False, "El monto no puede ser negativo."

    # 3. Creación con estado inicial
    incidente = Incidente(
        ...,
        id_alquiler=id_alquiler,
        tipo=tipo,
        monto=monto,
        estado="PENDIENTE" # Por defecto nace como deuda
    )

    IncidenteRepository.crear(incidente)
    return True, incidente
```

4.2. Gestión de Cobranza (Cambio de Estado)

Para que un incidente impacte en la facturación de la empresa, debe ser explícitamente cobrado. El sistema provee un método dedicado para transicionar el estado, evitando modificaciones accidentales.

Este diseño separa la detección del problema (ej: llega una multa por correo días después) de su resolución financiera (el cliente paga).

Fragmento de Código (src/services/incidente_service.py):

```
@staticmethod
def marcar_incidente_como_pagado(id_incidente):
    # Recuperamos la entidad
    incidente = IncidenteRepository.obtener_por_id(id_incidente)

    # Validamos lógica de negocio (no pagar dos veces)
    if incidente.estado == "PAGADO":
        return False, "El incidente ya se encuentra pagado."

    # Actualizamos estado
    incidente.estado = "PAGADO"
    IncidenteRepository.actualizar(incidente)

    return True, "Pago registrado exitosamente."
```

4.3. Impacto en Reportes Económicos

Aunque esta lógica reside en la capa de reportes, es importante documentar aquí que el diseño de la entidad Incidente fue pensado para integrarse con el balance.

El reporte "Resumen Económico" filtra explícitamente por el atributo estado definido en este módulo, sumando únicamente aquellos incidentes donde estado == 'PAGADO'.

Alcance 5: Emitir Reportes y Estadísticas

Requisito Propuesto: "Facilitar la emisión de reportes administrativos y estadísticos... Listado de alquileres por cliente, Vehículos más alquilados, Alquileres por mes o trimestre, Gráfico de facturación mensual."

Cumplimiento Técnico: El sistema implementa un módulo dedicado de Reportes que desacopla la lógica de extracción de datos (reportes_tablas.py), la visualización gráfica (reportes_graficos.py) y la exportación documental (reportes_export.py).

Utiliza librerías estándar de la industria (Matplotlib y ReportLab) para garantizar resultados profesionales.

5.1. Procesamiento de Datos (Lógica de Negocio)

Antes de graficar, los datos deben ser procesados. El sistema recupera la información cruda de la base de datos y aplica filtros de fecha y lógica de agrupación en memoria.

• **Filtrado Temporal:** Todos los reportes aceptan un rango de fechas (desde, hasta) y descartan los registros fuera de ese período.

• **Agregación:** Se utilizan diccionarios (defaultdict) para agrupar totales por vehículo, cliente o mes.

Fragmento de Código (src/reports/reportes_tablas.py):

```
def obtener_top_vehiculos(fecha_desde, fecha_hasta, limite=10):
    # 1. Recuperar todos los alquileres
    alquileres = AlquilerRepository.listar()
```

```
    # 2. Estructura para agrupar
    stats = defaultdict(lambda: {"cantidad": 0, "total": 0.0})

    for a in alquileres:
        # Filtro: Solo alquileres CERRADOS y dentro del rango
        if a.estado != "CERRADO": continue
        if not _en_rango(a.fecha_inicio, fecha_desde, fecha_hasta): continue

        # Agregación
        stats[a.id_vehiculo]["cantidad"] += 1
        stats[a.id_vehiculo]["total"] += a.total

    # 3. Ordenamiento y Corte
    filas.sort(key=lambda x: x["cantidad"], reverse=True)
    return True, filas[:limite]
```

5.2. Visualización Gráfica (Matplotlib)

Para los indicadores visuales, se generan gráficos de barras y de torta. Estos gráficos no se guardan en disco, sino que se renderizan dinámicamente en una ventana emergente dentro de la aplicación Tkinter.

Facturación Mensual: Gráfico de barras temporal.

Estado de Flota: Gráfico de torta (Pie Chart) mostrando la distribución porcentual.

Fragmento de Código (src/reports/reportes_graficos.py):

```
def grafico_facturacion_mensual(fecha_desde, fecha_hasta):
    # Obtener datos procesados
    ok, filas = rpt.obtener_alquileres_por_mes(fecha_desde, fecha_hasta)

    # Preparar datos para Matplotlib
    etiquetas = [f["periodo"] for f in filas] # Eje X: "2023-10", "2023-11"
    valores = [f["total"] for f in filas]     # Eje Y: Montos

    # Crear Figura
    fig = Figure(figsize=(8, 4))
    ax = fig.add_subplot(111)

    ax.bar(etiquetas, valores, color="#3A7AFE")
    ax.set_title(f"Facturación mensual {fecha_desde} a {fecha_hasta}")
    ax.set_ylabel("Monto ($)")

    return True, fig
```

5.3. Exportación a PDF (ReportLab)

El sistema permite generar documentos formales listos para imprimir. Se utiliza la librería ReportLab para "dibujar" el PDF vectorialmente, asegurando máxima calidad.

• **Diseño:** Se incluye un encabezado institucional, fecha de emisión y tablas con alineación monetaria.

• **Paginación:** El sistema controla automáticamente los saltos de página si el reporte es muy largo.

Fragmento de Código (src/reports/reportes_export.py):

```
def export_resumen_economico_pdf(ruta_pdf, fecha_desde, fecha_hasta):
    # Obtener datos
    ok, data = rpt.obtener_resumen_economico(fecha_desde, fecha_hasta)

    c = canvas.Canvas(ruta_pdf, pagesize=A4)

    # Encabezado
    c.setFont("Helvetica-Bold", 16)
    c.drawString(2 * cm, 27 * cm, "Resumen Económico")
    c.drawString(2 * cm, 26.2 * cm, f"Período: {fecha_desde} a {fecha_hasta}")
    c.line(2 * cm, 26 * cm, 19 * cm, 26 * cm) # Línea separadora

    # Cuerpo del reporte
    y = 25 * cm
    c.drawString(2 * cm, y, "Concepto")
    c.drawRightString(18 * cm, y, f"${data['total_general']:.2f}")

    c.save() # Generar archivo físico
    return True, "PDF generado correctamente."
```

Alcance 6: Manejo de Persistencia de Datos

Requisito Propuesto: "Manejo de persistencia de datos: Base de datos relacional con claves primarias y foráneas."

Cumplimiento Técnico: El sistema utiliza SQLite3 como motor de base de datos relacional. Esta elección permite que la aplicación sea portable (el archivo .db se crea localmente) sin requerir la instalación de servidores externos, ideal para una aplicación de escritorio.

Se implementó un esquema relacional estricto que garantiza la integridad de los datos mediante restricciones de claves primarias (PK) y claves foráneas (FK).

6.1. Integridad Referencial (Claves Foráneas)

SQLite, por defecto, no hace cumplir las restricciones de clave foránea. Para cumplir con el requisito académico y asegurar la consistencia (por ejemplo, no poder crear un alquiler para un cliente que no existe), el sistema activa explícitamente esta validación en cada conexión.

Fragmento de Código (src/repositories/db_connection.py):

```
def get_connection(self):
    if self._connection is None:
        self._connection = sqlite3.connect(DB_PATH)
        # ACTIVACIÓN OBLIGATORIA DE INTEGRIDAD REFERENCIAL
        self._connection.execute("PRAGMA foreign_keys = ON;")
    return self._connection
```

6.2. Diseño del Esquema Relacional

El script de inicialización (init_db) define la estructura de tablas. Se utilizan:

• **PRIMARY KEY AUTOINCREMENT:** Para identificadores únicos automáticos (IDs).

• **FOREIGN KEY:** Para relacionar entidades.

• **CONSTRAINTS:** Como NOT NULL o UNIQUE (ej: DNI y Patente) para evitar datos duplicados o incompletos a nivel de base de datos.

Ejemplo: Tabla alquileres (Relación Central)

Esta tabla actúa como el nexo principal del sistema, vinculando Cliente, Vehículo y Empleado.

Fragmento de Código (SQL en src/repositories/db_connection.py):

```
CREATE TABLE IF NOT EXISTS alquileres (
    id_alquiler    INTEGER PRIMARY KEY AUTOINCREMENT, -- PK
    id_cliente     INTEGER NOT NULL,
    id_vehiculo    INTEGER NOT NULL,
    id_empleado    INTEGER NOT NULL,
    fecha_inicio TEXT    NOT NULL,
    -- ... otros campos ...

    -- Definición de Claves Foráneas
    FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente),
```

```
    FOREIGN KEY (id_vehiculo) REFERENCES vehiculos(id_vehiculo),
    FOREIGN KEY (id_empleado) REFERENCES empleados(id_empleado)
);
```

6.3. Índices para Rendimiento

Para cumplir con el requisito no funcional de "Rendimiento básico", se crearon índices en las columnas más consultadas, específicamente aquellas usadas para filtrar reportes por fecha o buscar historiales.

Fragmento de Código (SQL):

```
-- Acelera el reporte "Resumen Económico" y la validación de disponibilidad
CREATE INDEX IF NOT EXISTS idx_alquileres_fecha_inicio
    ON alquileres (fecha_inicio);

-- Acelera la búsqueda de incidentes por alquiler
CREATE INDEX IF NOT EXISTS idx_incidentes_id_alquiler
    ON incidentes (id_alquiler);
```

Alcance 7: Interfaz de Usuario (UI/UX)

Requisito Propuesto: "Plataforma de escritorio... para interacción con formularios de carga, consulta y reportes. Usabilidad: formularios claros, validación de campos y mensajes descriptivos."

Cumplimiento Técnico: La interfaz gráfica fue desarrollada utilizando la librería CustomTkinter, una extensión moderna de Tkinter que permite crear aplicaciones con estética nativa, bordes redondeados y soporte automático para temas (Modo Claro/Oscuro).

Se adoptó un diseño modular basado en Frames intercambiables sobre un contenedor principal, simulando la navegación de una aplicación web pero en escritorio.

7.1. Diseño Moderno y Responsivo

El sistema abandona los controles grises estándar de Windows por componentes estilizados. Se implementó un Dashboard Principal que actúa como centro de mando, ofreciendo indicadores clave (KPIs) y accesos directos visuales.

Adaptabilidad: Se utiliza el gestor de geometría .grid() con pesos (weight) configurados en filas y columnas. Esto permite que, al maximizar la ventana, los elementos (como tablas y formularios) se expandan proporcionalmente para ocupar el espacio disponible.

Feedback Visual: Los botones cambian de color al pasar el mouse (hover effect), y los campos de entrada utilizan placeholders para guiar al usuario.

Fragmento de Código (src/ui/gui/app.py - Configuración Global):

```
# Configuración del tema visual para toda la aplicación
ctk.set_appearance_mode("System") # Detecta si el SO está en modo oscuro o claro
ctk.set_default_color_theme("blue") # Paleta de colores coherente
```

```
class App(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.title("Sistema de Alquiler de Vehículos")
        self.geometry("1100x700") # Resolución inicial optimizada
        # ...
```

7.2. Patrón de Diseño de Pantallas (Split View)

Para las pantallas de gestión (ABM), se estandarizó un diseño de Panel Dividido:

• **Izquierda (Fijo):** Formulario de carga de datos. Mantiene un ancho fijo para asegurar la legibilidad de los campos.

• **Derecha (Flexible):** Tabla de listado (Treeview). Se expande horizontalmente para mostrar todas las columnas necesarias sin cortar información.

Esta disposición mejora la usabilidad al permitir ver el listado completo mientras se edita o crea un registro nuevo, sin necesidad de abrir ventanas emergentes.

*Fragmento de Código (Estructura típica en *_screen.py):*

```
def _construir_ui(self):
    # Configuración de pesos para responsividad
    self.grid_columnconfigure(1, weight=1) # La columna 1 (Tabla) se expande

    # Panel Izquierdo: Formulario
    form_frame = ctk.CTkFrame(self, width=320)
    form_frame.grid(row=0, column=0, sticky="ns") # Pegado arriba y abajo

    # Panel Derecho: Tabla
    tabla_container = ctk.CTkFrame(self)
    tabla_container.grid(row=0, column=1, sticky="nsew") # Se expande en todas direcciones
```

7.3. Integración de Componentes Externos

Para cumplir con requisitos específicos que CustomTkinter no cubre nativamente, se integraron librerías de terceros de forma transparente:

• **TkCalendar:** Para la selección de fechas en alquileres y reportes, evitando errores de formato manual (dd/mm/aaaa).

• **Matplotlib Backend:** Los gráficos estadísticos se renderizan dentro de ventanas de Tkinter utilizando FigureCanvasTkAgg, permitiendo visualizar el análisis de datos sin salir de la aplicación.

■ Conclusión Final del Proyecto

El sistema RentaYa ha logrado cumplir satisfactoriamente con todos los objetivos planteados en la propuesta inicial.

• **Robustez:** La arquitectura en capas y el uso de Singleton en la base de datos garantizan un funcionamiento estable.

• **Flexibilidad:** El patrón Factory Method prepara el sistema para futuros tipos de vehículos sin reescribir código base.

• **Usabilidad:** La interfaz moderna facilita la operación diaria para usuarios no técnicos.

• **Integridad:** Las validaciones de negocio cruzadas (fechas, mantenimientos, estados) impiden errores operativos comunes en el rubro.

El software entregado constituye una solución funcional, escalable y profesional para la problemática planteada.

8. Requisitos No Funcionales (Calidad de Software)

Estos requisitos definen los criterios de calidad y las restricciones técnicas bajo las cuales opera el sistema.

8.1. Consistencia de Datos

Requisito: "Reglas de negocio centralizadas (disponibilidad, estados, cálculos)."

Cumplimiento Técnico: Para garantizar la consistencia, se prohibió el acceso directo a la base de datos desde la interfaz gráfica. Toda operación de escritura debe pasar obligatoriamente por la Capa de Servicio, que actúa como gatekeeper (guardián) de la integridad.

• **Unicidad de Reglas:** El cálculo del costo de alquiler, por ejemplo, existe en un solo lugar (AlquilerService). Si se cambia la regla de negocio, se actualiza allí y se replica en todo el sistema.

• **Integridad Referencial:** El motor de base de datos (SQLite) está configurado para rechazar operaciones que violen claves foráneas (ej: borrar un cliente que tiene alquileres activos).

Fragmento de Código (src/repositories/db_connection.py):

```
def get_connection(self):
    # ...
    # Forzamos la consistencia relacional a nivel de motor de BD
    self._connection.execute("PRAGMA foreign_keys = ON;")
    return self._connection
```

8.2. Rendimiento Básico

Requisito: "Uso de índices en campos de fechas y vehículo para consultas frecuentes."

Cumplimiento Técnico: Dado que los reportes y las validaciones de disponibilidad requieren filtrar por rangos de fechas y claves foráneas, se crearon índices B-Tree en la base de datos. Esto reduce

la complejidad de búsqueda de O(n) (escaneo completo de tabla) a O(log n).

Fragmento de Código (SQL en init_db):

```
-- Acelera el reporte "Resumen Económico" y la validación de disponibilidad
CREATE INDEX IF NOT EXISTS idx_alquileres_fecha_inicio
    ON alquileres (fecha_inicio);

-- Acelera la búsqueda de incidentes por alquiler
CREATE INDEX IF NOT EXISTS idx_incidentes_id_alquiler
    ON incidentes (id_alquiler);
```

8.3. Mantenibilidad

Requisito: "Estructura en capas y scripts de creación de base de datos."

Cumplimiento Técnico: La arquitectura del proyecto separa las responsabilidades en directorios claros (/ui, /services, /repositories, /domain). Esto permite que un cambio en la interfaz gráfica (ej: cambiar el color de un botón) no afecte la lógica de negocio ni el acceso a datos.

Además, el sistema es autocontenido: el script init_db() verifica la existencia de las tablas al inicio y aplica migraciones automáticas si detecta que faltan columnas, facilitando el despliegue en nuevas máquinas.

Estructura de Archivos:

```
/src
/domain     -> Definiciones de clases puras (Entidades)
/repositories -> Consultas SQL y conexión (Singleton)
/services     -> Lógica de negocio y validaciones (Factory)
/ui         -> Pantallas y componentes visuales (CustomTkinter)
```

8.4. Usabilidad

Requisito: "Formularios claros, validación de campos y mensajes descriptivos."

Cumplimiento Técnico: Se priorizó la experiencia de usuario (UX) mediante:

• **Feedback Inmediato:** Uso de messagebox.showwarning para errores de validación (ej: "DNI inválido") y showinfo para confirmaciones.

• **Prevención de Errores:** Uso de DateEntry para evitar errores de formato al escribir fechas y ComboBox de solo lectura (state="readonly") para evitar entradas inválidas en selecciones.

• **Diseño Limpio:** Los formularios utilizan placeholders (texto de ayuda dentro del campo) para reducir el ruido visual.

Fragmento de Código (src/ui/gui/clientes_screen.py):

```
# Ejemplo de Usabilidad: Placeholder y Validación previa
self.entry_dni = ctk.CTkEntry(form_frame, placeholder_text="DNI (Solo números)")

def _validar_form(self):
```

```
    # Mensaje descriptivo al usuario
    if not dni.isdigit():
        messagebox.showwarning("Validación", "El DNI debe contener solo números.")
        return None
```

Fin del Informe Técnico