PROYECTO TÉCNICO EXPLORADOR DE TASACIONES INMOBILIARIAS EN PYTHON

Diciembre de 2019

Contenido

1. Introducción

El siguiente trabajo comprende la creación de un programa sencillo en Python que permite guardar y editar información de tasaciones de propiedades en una base de datos e imágenes de las mismas en un directorio local. El programa presenta distintas opciones de visualización e interacción de los datos guardados.

2. Presentación del programa

Con la herramienta gráfica tkinter se crea una ventana principal que funciona como explorador de las tasaciones donde se pueden visualizar los datos alojados en la base de datos y las imágenes guardadas en el directorio local.

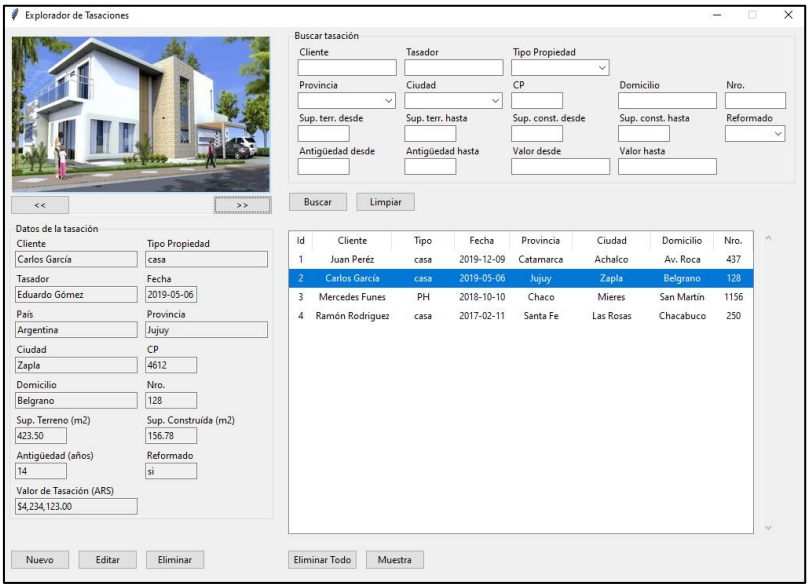

Figura 1 – Ventana Principal

Cuando se quiere agregar un nuevo informe de tasación, o bien editar uno ya existente, se abre una ventana emergente (objeto Toplevel()) que permite el ingreso de los datos y la carga de imágenes.

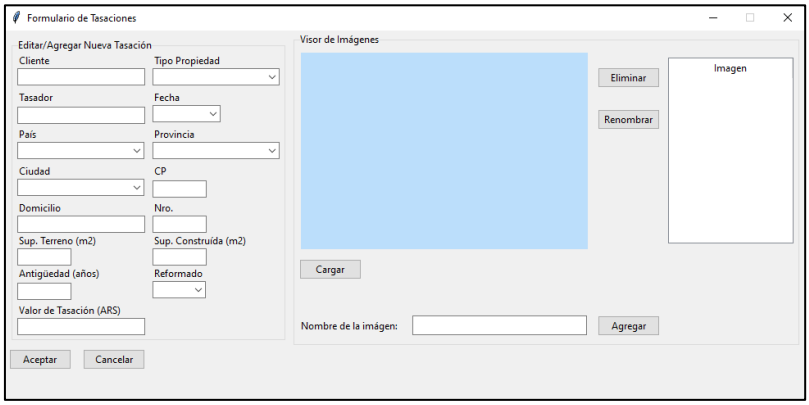

Figura 2 - Formulario (toplevel)

3. Estructura del programa

La siguiente figura muestra un esquema de los archivos y directorios que componen el programa y la forma en que se relacionan. Si bien se va a profundizar más adelante, en “main.py” se encuentra la clase que genera la ventana principal del programa y en “formulario.py” la ventana emergente o formulario que se lanza para la creación o edición de los registros. La idea es que las consultas a la base de datos se gestionen desde “gestionar.py”. Desde aquí también se conecta a un archivo “localidades.json” que alberga datos de interés. Tanto la ventana principal como el formulario (ventana emergente) requieren conectarse a la base de datos para devolver información al usuario a través de tablas. Además, acceden a un directorio local donde guardan o abren las imágenes asignadas a las tasaciones.

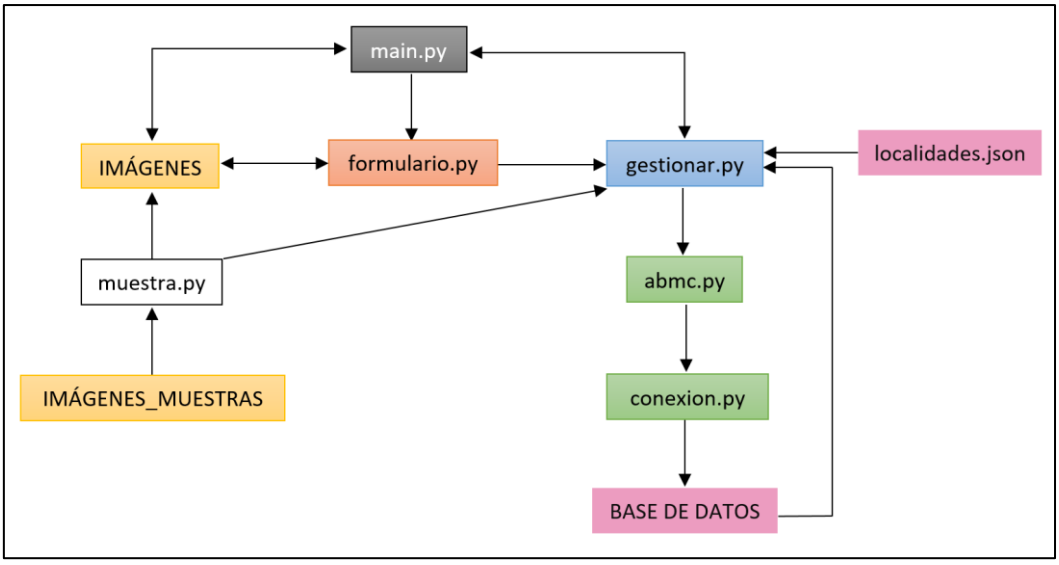

Figura 3 - Estructura del programa

El directorio del programa queda compuesto entonces por:

```
data
        localidades.json
documentación
        manual_tecnico.pdf
img
        readme
        imagenes_muestras
                viv_CZ_01.jpg
                …
src
        abmc.py
        conexión.py
        formulario.py
        gestionar.py
        muestra.py
main.py
README.md
requirements.txt
```

A continuación, se desarrolla el contenido de los módulos “main.py” y “formulario.py” desde donde se verá su integración con los demás módulos del programa y los directorios de trabajo.

4. Módulo “main.py”

4.1. Interfaz gráfica e inicio

Este módulo contiene la clase *“Aplicacion”* a través de la cual se crea la ventana principal del programa. Gráficamente se divide en dos grandes secciones, una donde se visualizan los datos de las tasaciones (sección de áreas de datos) y la otra donde se disponen los botones principales (sección de botones principales). La sección de datos a su vez se divide en cuatro áreas:

- Área tabla de clientes: cada informe de tasación corresponde a un cliente el cual se puede visualizar de forma rápida junto con otros datos resumidos en una tabla.
- Área imagen: es el lugar dónde se visualizan las imágenes guardadas en el directorio correspondientes a la tasación seleccionada en la tabla de clientes. Cuenta con dos botones que permiten pasar a la imagen anterior / siguiente.

- Área buscar / filtrar: cuenta con entradas y listas desplegables para ingresar, o seleccionar, texto o valores numéricos que ayuden a filtrar las filas de la tabla de clientes cuando se ejecuta el botón “Buscar”. Además, tiene otro botón “Limpiar” que borra el texto ingresado para realizar nuevas búsquedas.
- Área información de la tasación: cuando se selecciona una fila en la tabla de clientes se visualizan en el área todas las características de la tasación almacenadas en la base de datos.

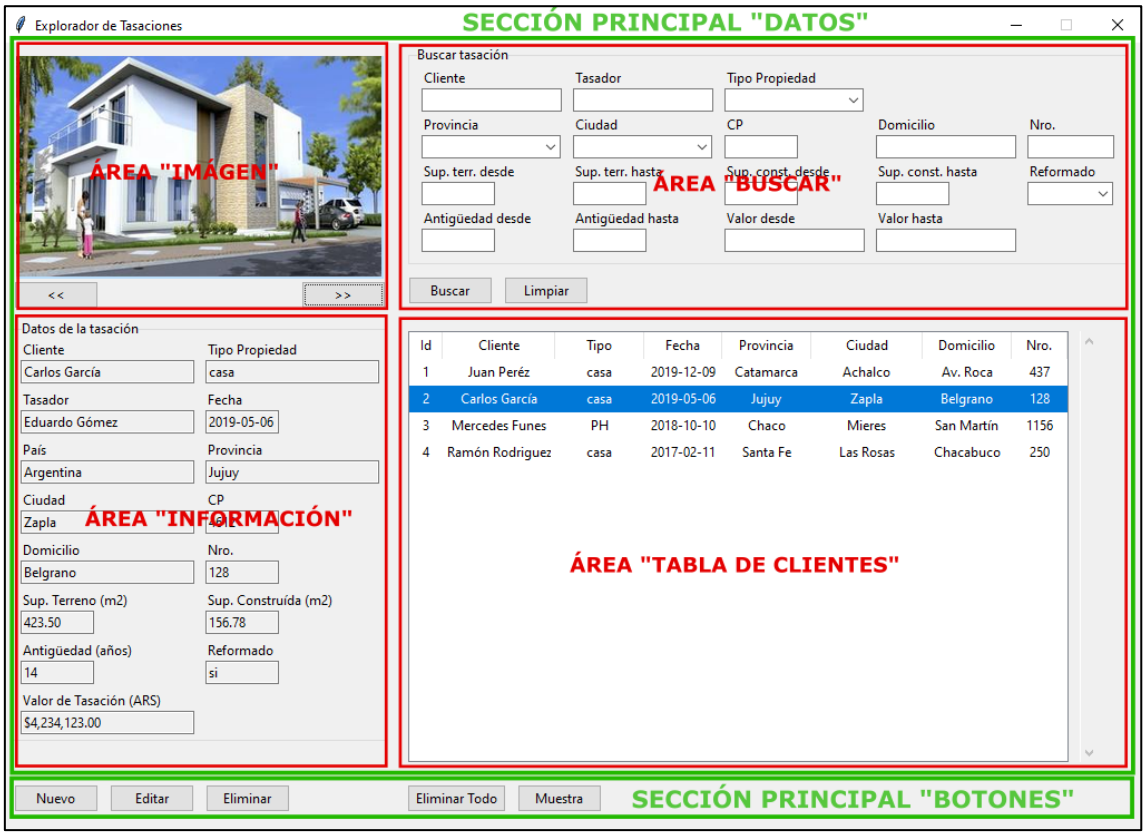

Figura 4 - Interfaz gráfica de la ventana principal

main.py

```
def __create_widgets(self):
        """
        Se crean los widgets de la clase.
        """
        # CONTENEDOR PRINCIPAL
        # ---------------------------------------------------------------------------------------------
        self.contenedor = ttk.Frame(self.parent, style = "test.TFrame")
        self.contenedor.pack(expand="yes", fill=BOTH)

        # SECCIONES PRINCIPALES
        # ---------------------------------------------------------------------------------------------
        # Sección de áreas de datos
        self.seccion_datos = ttk.Frame(self.contenedor,style = "test.TFrame")
        self.seccion_datos.pack(side=TOP, expand="yes", fill=BOTH, padx= 0, pady= 0)

        # Sección de botones
        self.seccion_botones = ttk.Frame(self.contenedor)
        self.seccion_botones.pack(side = TOP, expand = "yes", fill = BOTH, padx = 10, pady = 10)

        # ÁREA IMÁGEN
        # ---------------------------------------------------------------------------------------------
        # Contenedor del área
        self.area_imagen = ttk.Frame(self.seccion_datos, style = "area_imagen.TFrame")
        self.area_imagen.grid(row = 0, column = 1, padx = 10, pady =  10, sticky= N )
        . . . . . .

        # ÁREA BUSCAR / FILTRAR
        # ---------------------------------------------------------------------------------------------
```

```
        # Contenedor del área
        self.area_buscar = ttk.Frame(self.seccion_datos)
        self.area_buscar.grid(row = 0, column = 2, padx = 10, pady = 0, sticky = W)
        . . . . . .

        # ACCIONES EN LOS COMBOBOXS
        # Se captura la provincia seleccionada para filtrar las localidades
        self.cmbbox_provincia.bind('<<ComboboxSelected>>',
                                   lambda event:get_localidad(self.cmbbox_ciudad,
                                   self.cmbbox_provincia))
        # Se captura la ciudad seleccionada para llenar campo "CP"
        self.cmbbox_ciudad.bind('<<ComboboxSelected>>',
                                lambda event:set_cp(self.cmbbox_ciudad, self.entry_buscar[2]))


        # ÁREA INFORMACIÓN DE LA TASACIÓN
        # ------------------------------------------------------------------------------------------
        # Contenedor del área
        self.area_informacion = ttk.Frame(self.seccion_datos)
        self.area_informacion.grid(row = 1, column = 1, sticky = N)
        . . . . . .

        # ÁREA TABLA DE CLIENTES
        # ------------------------------------------------------------------------------------------
        # Contenedor del área
        self.area_tabla = ttk.Frame(self.seccion_datos, style = "test.TFrame")
        self.area_tabla.grid(row = 1 , column = 2, sticky = N, padx = (0,100) , pady = 10)
        . . . . . .

        # ACCIONES EN LA TABLA
        # Se llena la tabla con los datos (tasaciones) existentes en la bd
        self.vista_tasacion = vista_tasacion(self.tabla)
        # Al seleccionar una fila se llenan las entradas del área de información de la tasación
        self.tabla.bind('<<TreeviewSelect>>', self.on_select)

        # ÁREA BOTONES PRINCIPALES
        # ------------------------------------------------------------------------------------------
        # Botones
        self.btn_nuevo = Button(self.seccion_botones, text = "  Nuevo  ",
                                command = self.nuevo_formulario)
        self.btn_nuevo.grid(row = 0, column = 1, sticky = W + E, padx = (0,10), pady = (0,10))
        . . . . . .

        # CREAR UNA MUESTRA PARA PROBAR EL PROGRAMA
        # ------------------------------------------------------------------------------------------
        # Se agrega un botón adicional para crear una muestra de registros (tasaciones) predefinidas
        self.btn_muestra = Button(self.seccion_botones, text = "  Muestra  ",
                                  command = lambda:crear_muestra(self.tabla))
        self.btn_muestra.grid(row = 0, column = 5, sticky = E, padx = (10,10), pady = (0,10))

        # ACCIONES EN LOS BOTONES
        # Si no hay acceso a la bd los botones se desactivan
        self.estado_botones()
```

Cuando se ejecuta el programa, es decir, cuando se corre el archivo “main.py” se genera la interfaz gráfica de la ventana principal; luego de crear la tabla de clientes (“self.tabla”) se acciona la función **vista_tasación()**, ubicada en gestionar.py, que intenta conectar a la base de datos para obtener los datos guardados en la misma y volcarlos a la tabla. Si MySQL no está activo se emite un mensaje de error, la función **vista_tasación()** devuelve un valor “None”, y el método **estado_botones()** desactiva los botones de la ventana principal para que no se pueda realizar ninguna acción en ella.

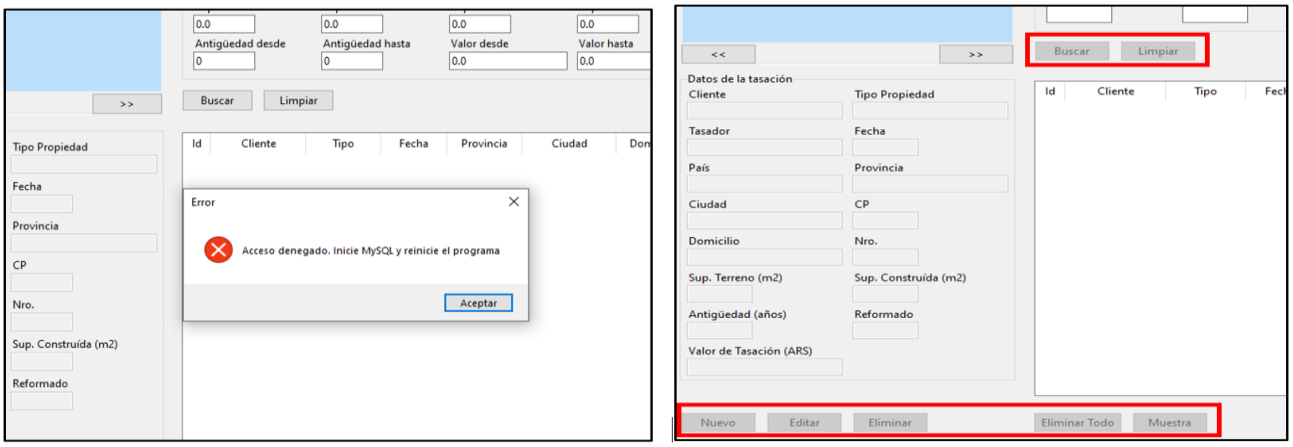

Figura 5 - Mensaje de error y botones desactivados cuando falla la conexión a MySQL

Si MySQL está conectado y no existe la base de datos solicitada ésta se crea, junto con una tabla definida, y la aplicación se abre correctamente mostrando la tabla de clientes vacía. En tanto que, si hay una base de datos y una tabla con registros de tasaciones, éstos se inyectan en la tabla de clientes para visualizar parte de la información.

Concretamente, la función **vista_tasacion()** intenta conectar a la tabla de una base de datos solicitada para recuperar la información alojada a través de la instanciación del método **consulta()** de la clase **Abmc()**. Si la conexión es exitosa y hay datos disponibles éstos se insertan en la tabla de clientes (pasada como argumento "tabla") por medio del método tabla.**insert()** de ttk.

gestionar.py

```
# Visualizar cambios en tabla
def vista_tasacion(tabla):
    """
    Muestra los registros de la bd en la tabla de la ventana principal. Es decir, se conecta a la
    bd, se recuperan los valores de la tabla "registros" y se insertan en la tabla de la ventana
    de inicio del programa.
    """
    # Se trata de conectar y consultar los registros de la tabla en la bd
    db_consulta = Abmc().consulta()
    # Si hay conexión se ejecuta el código
    if  isinstance(db_consulta, list):
        # Se eliminan (limpian) las filas existentes en la tabla del programa
        records = tabla.get_children()
        for element in records:
            tabla.delete(element)
        # Se llena la tabla con las filas actualizadas
        for row in db_consulta:
            tabla.insert('', "end", text = row, values = row)
        return True
    else:
        # Cuando falla la conexión a la tabla de la bd
        return None
```

La instancia **Abmc().consulta()** entonces tratará de conectarse a la tabla llamada "registros" en la base de datos "tasaciones" (los nombres ya están definidos internamente). Si efectivamente la base de datos y la tabla existen y se obtiene acceso se ejecuta la consulta **'SELECT * FROM registros'**, de lo contrario como ya se explicó, **vista_tasacion()** retornará el valor "None".

abmc.py

```
def consulta(self):
        """
        Si hay conexión a la tabla "registros" de la bd "tasaciones" devuelve los valores de la
        tabla.
        """
        if self.conexion:
            # Trata de recuperar los valores de la tabla
            try:
```

```
                cursor = self.conexion.cursor()
                # Se consulta la bd
                sql = 'SELECT * FROM registros'
                cursor.execute(sql)
                db_rows = cursor.fetchall()
                return db_rows
            except:
                messagebox.showwarning("Alerta", "No se puede realizar la consulta")
            # Se cierra la conexión
            finally:
                if self.conexion:
                    self.conexion.close()
        else:
            return None
```

La clase “Abmc” hereda de la clase padre “Basededatos” para poder efectuar las operaciones de conexión con MySQL. Cuando se instancia la clase se ejecuta el método **conectar_bd()** el cual primero intenta conectar a la base de datos “tasaciones”, si no puede verifica que el servidor MySQL está activado para luego (si hay acceso) crear la base de datos “tasaciones” asumiendo que no existe (lo más probable si llega a este punto) y seguidamente la tabla “registros”. Finalmente, intenta nuevamente establecer conexión con la base de datos.

abmc.py

```
class Abmc(Basededatos):
    """
    La clase permite realizar operaciones básicas en la base de datos. Hereda como padre de la
    clase Basededatos para acceder a los métodos de conexión.
    """

    def __init__(self):
        """
        Se intenta abrir la conexión a la tabla "registros" de la bd.
        """
        self.conexion = self.conectar_bd()
. . . . . .
```

conexion.py

```
def conectar_bd(self):
            """
            En primer lugar se trata de conectar a la bd "tasaciones". Si no es posible la conexión
            se verifica que MySQL esté en funcionamiento, de ser así, significa que no existe la bd
            y se crea junto con la tabla "registros". Por último, se intenta establecer nuevamente la
            conexión a la bd "tasaciones".
            """
            # Se trata de conectar a la bd
            try:
                conexion = mysql.connector.connect(host = "localhost", user = "root", passwd = "",
                                                   database = "tasaciones")
                return conexion
            except:
                # Se verifica el acceso a MySQL
                conexion = self.acceso_mysql()
                if conexion:
                    cursor = conexion.cursor()
                    # Si hay acceso a MySQL crea la bd "tasaciones"
                    cursor.execute("CREATE DATABASE IF NOT EXISTS tasaciones")
                    conexion.close()
                    # Se crea la tabla "registros" en la bd
                    self.crear_tabla()
                    # Se vuelve a intentar la conexión con la bd
                    return self.conectar_bd()
```

Lo explicado hasta el momento sucede inmediatamente al iniciar el programa. Dentro de él se pueden realizar además diversas acciones o eventos en cada una de las secciones y áreas mencionadas.

4.2. Área tabla de clientes

La tabla se construye con un widget Treeview del paquete Tkinter.ttk. En ella se muestran algunas columnas del total de datos disponibles a modo de resumen (Cliente, Tipo de Propiedad, Fecha, etc.). El resto de las columnas permanecen ocultas, pero se puede acceder a los datos que contienen. Se complementa la tabla de clientes con un widget Scrollbar vertical que se activa cuando el número de filas (tasaciones) excede la altura fijada para el objeto.

main.py

```
# ÁREA TABLA DE CLIENTES
# ------------------------------------------------------------------------------------------
# Contenedor del área
self.area_tabla = ttk.Frame(self.seccion_datos, style = "test.TFrame")
self.area_tabla.grid(row = 1 , column = 2, sticky = N, padx = (0,100) , pady = 10)

# Lista con todas las columnas de la bd
cols_tabla = ["Id","Cliente", "Tipo", "Tasador", "Fecha", "País", "Provincia", "Ciudad",
              "CP", "Domicilio", "Nro.", "Sup. terreno", "Sup. construida", "Antigüedad",
              "Reformado", "Valuación", "imagenes"]
 # Lista de las columnas que se van a mostrar en la tabla
 cols_mostrar = ["Id", "Cliente", "Tipo", "Fecha", "Provincia", "Ciudad", "Domicilio", "Nro."]
 # Anchos de las columnas
 width_cols = [30, 100, 80, 1, 70, 1, 80, 100, 1, 80, 50, 1, 1, 1, 1, 1, 1]
 # Se crea la tabla con todas las columnas.
 # Se limita la selección en la tabla a una sola fila (selectmode = "browse")
 self.tabla = ttk.Treeview(self.area_tabla, columns = cols_tabla, height = 15,
                           show = "headings", selectmode = "browse", style = "tabla.Treeview")
 # Se muestran solamente las columnas elegidas
 self.tabla["displaycolumns"] = cols_mostrar
 # Se crean las columnas en la tabla
 for col in range(len(cols_tabla)):
     self.tabla.heading(cols_tabla[col], text = cols_tabla[col])
     self.tabla.column(cols_tabla[col], anchor = "center", width = width_cols[col])
 self.tabla.grid(padx = 10)

 # Scrollbar vertical para la tabla
 ysb = Scrollbar(self.area_tabla, orient = "vertical", command = self.tabla.yview)
 ysb.grid(row = 0, column = 1, sticky = "ns")
 self.tabla.configure(yscroll=ysb.set)

 # ACCIONES EN LA TABLA
 # Se llena la tabla con los datos (tasaciones) existentes en la bd
 self.vista_tasacion = vista_tasacion(self.tabla)
 # Al seleccionar una fila se llenan las entradas del área de información de la tasación
 self.tabla.bind('<<TreeviewSelect>>', self.on_select)
```

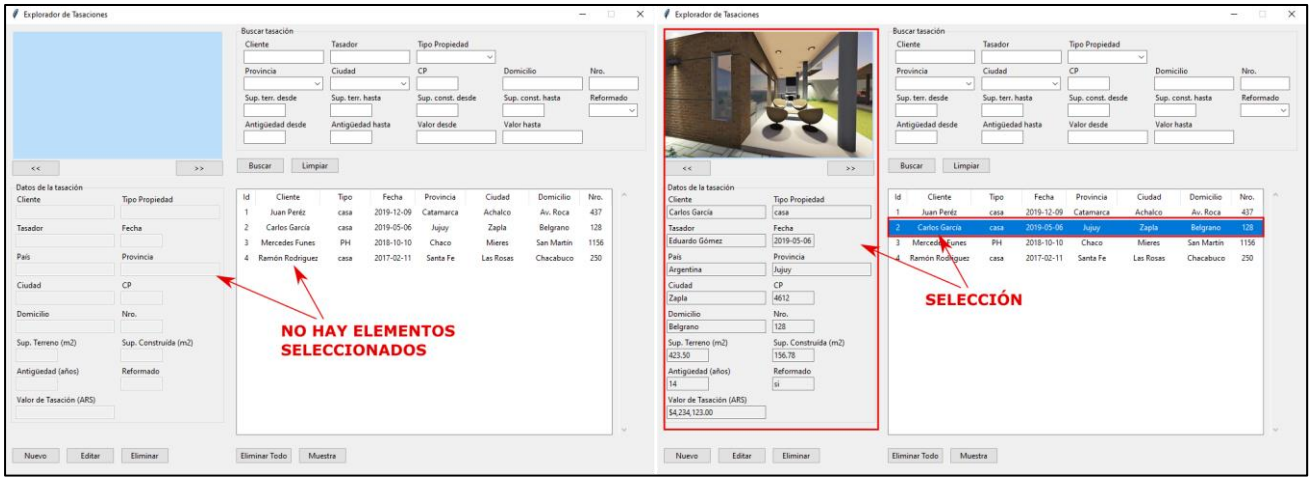

Ilustración 1 - Eventos al seleccionar una fila en la tabla de clientes

Se establece en la configuración la restricción de poder seleccionar una sola fila a la vez (no se pueden seleccionar múltiples filas) para evitar errores de acuerdo a como se diseñó el programa. Cuando se selecciona una

fila ocurre un evento, **on_select()**, que se encarga de llenar los campos de las entradas en el área de información de la tasación y mostrar las imágenes guardadas en el directorio pertenecientes al registro escogido en el área de imágenes.

En el interior de **on_select()**, el método treeview**.item()** de ttk permite recoger los valores de la fila seleccionada (con **"[values]"**) en una lista ("self.lista_items") que se recorre y cada elemento (salvo el último) se inserta en las entradas previstas para el mismo con el método entry**.insert()**.

main.py

```
def on_select(self, event):
        """
        Cuando se selecciona una fila en la tabla se recupera del widget Treeview toda la información
        de la tasación correspondiente al cliente (incluídos los datos de las columnas ocultas en la
        tabla) la cual se utiliza para completar los campos de las entradas del área de información
        y mostrar las imágenes guardadas en el directorio en el área de imágenes.
        """
        # Valores de la fila seleccionada (lista con datos de la tasación)
        self.seleccion_item_app = self.tabla.selection()
        self.lista_items = self.tabla.item(self.seleccion_item_app)["values"]
        # El último item que corresponde a los nombres de las imágenes guardades en el directorio
        self.string_imagenes = self.lista_items[-1]
        # Se transforma el string en una lista de nombres de imágenes
        self.lista_imagenes  = [i[1:-1] for i in list(self.string_imagenes[1:-1].split(", "))]
        # Se define la ubicación de las imágenes en el directorio
        self.ruta_imagenes   = [self.directorio_img + i for i in self.lista_imagenes]
        # Se llenan los campos del área de información con los valores de la selección
        for v in range(len(self.entry_informacion)-1):
            # Se cambia el estado de las entradas
            self.entry_informacion[v].configure(state='normal')
            self.entry_informacion[v].delete(0, END)
            self.entry_informacion[v].insert(0, self.lista_items[v+1])
            # Se cambia nuevamente el estado de las entradas
            self.entry_informacion[v].configure(state='readonly')
        # Se cambia el formato del valor de tasación
        self.entry_informacion[14].configure(state='normal')
        self.entry_informacion[14].delete(0, END)
        self.entry_informacion[14].insert(0, "${:,.2f}".format(float(self.lista_items[15])))
        self.entry_informacion[14].configure(state='readonly')
        # La función vista_imagenes muestra la primer imágen guardada en el directorio (si hubiera)
        self.vista_imagen(self.directorio_img +  self.lista_imagenes[0], (330,200))
        # Cada vez que se selecciona una fila en la tabla el contador vuelve a cero
        self.cnt = 0
```

El último elemento de la lista recuperada es un texto (string) con forma de lista que tiene nombres de imágenes. Se vuelve entonces a transformar el string a una estructura de lista y a cada elemento se le concatena la ubicación del directorio donde se guardan las imágenes obteniendo así una lista con las rutas de las imágenes de la tasación seleccionada ("self.ruta_imagenes"). El primer elemento de esa lista se proyecta en el canvas del área de imágenes por medio del método **vista_imagen()**. Dicho método utiliza la librería Pillow para cargar las imágenes e insertarlas en el canvas previsto con la configuración deseada.

main.py

```
def vista_imagen(self, open, size):
        """
        Este método carga imágenes desde el directorio y las muestra en un canvas, cuando se carga
        una lista vacía (no hay imágenes para mostar) se elimina la imágen previa cargada en el
        canvas correspondiente para que éste quede vacío.
        """
        try:
            # Se carga el archivo de imagen
            with PILImage.open(open) as abrir_img:
                resize_img = abrir_img.resize(size, PILImage.ANTIALIAS)
                self.imagen = ImageTk.PhotoImage(resize_img)
                self.canvas_app.create_image(0, 0, image = self.imagen, anchor='nw')
        except:
            # Se elimina la imagen existente en el canvas
            self.canvas_app.delete("all")
```

4.3. Área de imágenes

Básicamente se compone de un canvas donde se proyecta una imagen de una tasación seleccionada en la tabla de clientes y dos botones que permiten avanzar o retroceder hacia otra imagen de la misma tasación (si hubieran).

Antes se comentó que, de la tabla de clientes, haciendo algunas transformaciones se obtiene una lista con las rutas de las imágenes guardadas en el directorio de la tasación seleccionada. Y que además en el canvas se muestra la primera imagen de esa lista. Entonces, por ejemplo, para visualizar la siguiente imagen (suponiendo que exista) se ejecuta el botón “>>” que acciona el método **siguiente()** el cual captura el segundo elemento de la lista (ruta del archivo) y llama al método **vista_imagen()** para mostrar la imagen contigua. Para ir marcando la posición del siguiente o anterior elemento de la lista se usa un contador (se suma o resta 1 según el caso) el cual se reinicia (self.cnt = 0) al seleccionar otra tasación en la tabla cuando se dispara **on_select()**.

main.py

```
def siguiente(self):
        """
        Muestra la imágen siguiente de la lista de imágenes guardadas en el directorio
        pertenecientes a una tasación (o a un cliente) seleccionada en la tabla. Si la tasación no
        tiene imágenes guardadas, si hay una sola imágen o es la última imágen de la lista no se
        genera ningún cambio en el canvas. Se utiliza un contador para indicar la posición en
        la lista.
        """
        try:
            # Mientras no sea la última imagen de la lista
            if self.cnt < len(self.lista_imagenes) - 1:
                self.cnt += 1
            # Se carga la imagen al canvas
            self.vista_imagen(self.directorio_img + self.lista_imagenes[self.cnt], (330,200))
        except:
            # Es necesario seleccionar una fila en la tabla
             messagebox.showinfo("Alerta", "Seleccione una Tasación")

    def anterior(self):
        """
        Muestra la imágen anterior de la lista de imágenes guardadas en el directorio
        pertenecientes a una tasación (o a un cliente) seleccionada en la tabla. Si la tasación no
        tiene imágenes guardadas, si hay una sola imágen o es la primera imágen de la lista no se
        genera ningún cambio en el canvas. Se utiliza un contador para indicar la posición en
        la lista.
        """
        try:
            if self.cnt > 0:
                self.cnt -= 1
            self.vista_imagen(self.directorio_img + self.lista_imagenes[self.cnt], (330,200))
        except:
            messagebox.showinfo("Alerta", "Seleccione una Tasación")
```

4.4. Área de información de la tasación

El área comprende widgets de entradas que se utilizan para mostrar los datos cargados en una tasación al seleccionarla en la tabla de clientes. Ya se explicó que los datos se obtienen con treeview**.item()[“values”]** y se insertan en las respectivas entradas con entry**.insert()**. Las entradas se configuran con un estado ”readonly” (solo lectura) el cual cambia a "normal” antes de insertar los datos y después vuelve al estado definido inicialmente; la idea es que no se puedan modificar los campos desde la ventana de inicio.

4.5. Área de búsqueda

El área de búsqueda o filtrado se compone de elementos de entradas y de tipo combobox que permiten ingresar cadenas de texto o valores numéricos para filtrar las filas de la tabla de clientes cuando se aprieta el botón “Buscar” que ejecuta el método **buscar()**.

En el proyecto se incluye el archivo “localidades.json” que cuenta con registros de las provincias, ciudades y códigos postales de Argentina. Éste se carga en la variable “localidades” del módulo “gestionar.py”. Los elementos “self.cmbbox_provincia” de las clases “Aplicación” y “Formulario” extraen las provincias de dicha variable filtrando por “prv_nombre”. Al seleccionar en la lista desplegable una provincia se ejecuta un evento,

get_localidad(), el cual consiste en filtrar las ciudades de la variable “localidades” pertenecientes a la provincia elegida, e incorporar el resultado al widget “self.cmbbox_ciudad”. Por otro lado, al seleccionar una ciudad en dicho widget se manifiesta el evento **set_cp()** que inserta (también filtrando los datos) el valor del código postal de la ciudad en la entrada asignada.

main.py

```
. . . . . .

self.cmbbox_provincia["values"] = sorted(list(set([x['prv_nombre'] for x in localidades])))
self.cmbbox_provincia.grid(row = 4 , column = 1, padx = (10,10), pady = (0, 0), sticky = W)

self.cmbbox_ciudad = ttk.Combobox(self.marco_buscar, textvariable = StringVar(), width = 17)
self.cmbbox_ciudad["values"] = [""]
self.cmbbox_ciudad.grid(row = 4 , column = 2, padx = (0,10), pady = (0, 0), sticky = W)

. . . . . .

# ACCIONES EN LOS COMBOBOXS
# Se captura la provincia seleccionada para filtrar las localidades
self.cmbbox_provincia.bind('<<ComboboxSelected>>',
                           lambda event:get_localidad(self.cmbbox_ciudad,
                           self.cmbbox_provincia))
# Se captura la ciudad seleccionada para llenar campo "CP"
self.cmbbox_ciudad.bind('<<ComboboxSelected>>',
                        lambda event:set_cp(self.cmbbox_ciudad, self.entry_buscar[2]))

. . . . . .
```

gestionar.py

```
def get_localidad(ciudad, provincia):
    """
    Se filtran las localidades de acuerdo a la provincia seleccionada.
    """
    ciudad["values"] = sorted(list(set([x["loc_nombre"] for x in localidades
                       if x["prv_nombre"] == provincia.get()])))

def set_cp(ciudad, entrada_cp):
    """
    Se inserta en la entrada "CP" el código postal de la ciudad elegida.
    Si hay varias opciones cp para una ciudad, se elige la primera opción.
    """
    cp = [x["loc_cpostal"] for x in localidades if x["loc_nombre"] == ciudad.get()]
    entrada_cp.delete(0, END)
    entrada_cp.insert(0, cp[0])
```

El filtrado de las filas en la tabla de clientes se efectúa por un proceso de iteración que inicia el método **buscar()**. Las entradas disponibles permiten ingresar tanto texto como valores numéricos, por eso se comienza el filtrado (iteración) por un tipo y luego se pasa el siguiente tipo. Se comienza seleccionando una columna de la tabla, por ejemplo, la columna “Cliente”; se verifica en cada fila de la columna si el valor de la celda coincide con el texto ingresado en la entrada “Cliente” de la búsqueda (en minúscula) y si no es así se elimina la fila completa, o sea, el registro completo con todas sus columnas. Luego, con las filas que quedan sin eliminar se pasa a otra columna, por ejemplo, “Tasador” y así sucesivamente con cada columna se van eliminando los registros que no coinciden con los valores ingresados. Para el caso de los valores numéricos el filtrado puede hacerse entre un rango de valores ingresados.

main.py

```
def buscar(self):
        """
        Se selecciona una columna de la tabla de tasaciones con todas sus filas. En cada cada fila
        se evalúa la condición establecida en la entrada perteneciente a la variable elegida y si no
        cumple dicha condición se elimina la fila completa (todas las columnas) de la tabla. Cuando
        se verifican todos las filas de una columna se pasa a la siguiente columna donde se evalúan
```

```
        los registros que no se eliminaron y así susecivamente se van filtrando las filas.
        La condición puede incluir texto o valores numéricos.
        """
        # Se llena la tabla con los datos de la bd
        vista_tasacion(self.tabla)
        # Lista de indices de columnas con valores de tipo string en la tabla
        cols_string  = [1, 3, 2, 6, 7, 8, 9, 10, 14]
        # Lista de entradas correspondientes a las columnas e tipo string en el área buscar
        entry_string = [self.entry_buscar[0], self.entry_buscar[1], self.cmbbox_tipo,
                        self.cmbbox_provincia, self.cmbbox_ciudad, self.entry_buscar[2],
                        self.entry_buscar[3], self.entry_buscar[4], self.cmbbox_reformado]
        # Lista de índices de columnas con valores numéricos en la tabla
        cols_num  = [11, 12, 13, 15]
        # Lista de entradas correspondientes a las columnas numéricas en el área buscar
        entry_num = [[self.entry_buscar[5], self.entry_buscar[6]], [self.entry_buscar[7],
                     self.entry_buscar[8]], [self.entry_buscar[9], self.entry_buscar[10]],
                     [self.entry_buscar[11], self.entry_buscar[12]]]
        # Se filtran las filas si se ingresa texto
        for col in range(len(cols_string)):
            # Toma todas las filas de la tabla existentes para la columna "col"
            children = self.tabla.get_children()
            for child in children:
                # Toma una fila de la columna, la convierte en string y minúscula
                texto = str(list(self.tabla.item(child)["values"])[cols_string[col]]).lower()
                # Si no hay coincidencia con la búsqueda se elimina la fila completa
                if not texto.startswith(entry_string[col].get().lower()):
                    self.tabla.delete(child)
        # Se filtran las filas cuando se ingresan valores numéricos
        for col in range(len(cols_num)):
        # Para cada columna se toman todas las filas
            children = self.tabla.get_children()
            for child in children:
                # Se convierten los valores a tipo "float"
                valor = float(list(self.tabla.item(child)["values"])[cols_num[col]])
                # Si no hay coincidencia con la búsqueda se elimina la fila
                # Si se ingresa solo en campo "desde"
                if entry_num[col][0].get() != "" and entry_num[col][1].get() == "":
                    if valor < float(entry_num[col][0].get()):
                        self.tabla.delete(child)
                # Si se ingresa solo en campo "hasta"
                elif entry_num[col][0].get() == "" and entry_num[col][1].get() != "":
                    if valor > float(entry_num[col][1].get()):
                        self.tabla.delete(child)
                # Si se ingresan en campos "desde" y "hasta"
                elif entry_num[col][0].get() != "" and entry_num[col][1].get() != "":
                    if valor < float(entry_num[col][0].get()) or valor > float(entry_num[col][1].get()):
                        self.tabla.delete(child)
```

4.6. Botones principales

4.6.1. Botón “Muestra”

En la sección de botones principales se encuentra el botón “Muestra” que se incluye solamente para crear una muestra de registros de tasaciones para testear el programa, pero no se lo trata como parte de él por ello no se desarrolla su funcionamiento. Simplemente se menciona que el botón ejecuta una función que se ubica en el archivo “muestra.py” la cual graba cuatro tasaciones en la base de datos y le asigna a cada una de ellas unas imágenes preseleccionadas extraídas de la carpeta “imagenes_muestras”. Es posible ejecutar el botón varias veces, aunque los datos insertados se repetirán.

4.6.2. Botón “Eliminar”

Si se desea eliminar una tasación de la base de datos basta con seleccionarla en la tabla de clientes y apretar “Eliminar” para ejecutar el método **eliminar_fila()**.

main.py

```
def eliminar_fila(self):
        """
        Al seleccionar una fila y ejecutar el botón "Eliminar" se elimina el registro de la tabla en
        la bd. El método llama a la función "eliminar_registro" del módulo "gestionar.py" la cual se
        conecta a la bd a través de la clase "Abmc" y realiza la tarea requerida.
```

```
        """
        try:
            # Se verifica que se haya seleccionado una fila
            if self.seleccion_item_app:
                resultado =  messagebox.askquestion("Eliminar",
                                                    "¿Está seguro que desea eliminar el registro?")
                if resultado == "yes":
                    # Se elimina la fila de la tabla en la base de datos
                    # Se pasan como argumentos la tabla de tasaciones con el id de la que se debe eliminar
                    eliminar_registro(self.tabla, self.lista_items[0])
                    # Se elimina la variable para que no exista si la tabla está vacía y evitar un error
                    del self.seleccion_item_app
                    # Se eliminan las imágenes correspondientes en el directorio de trabajo (si hubiese)
                    if self.ruta_imagenes != [self.directorio_img]:
                        for imagen in sorted(self.ruta_imagenes, reverse = True):
                                os.remove(imagen)
        except:
           messagebox.showinfo("Alerta", "Seleccione una fila")
```

Dentro de **eliminar_fila()** se ejecuta la función **eliminar_registro()** donde es necesario pasar como argumento el id de la tasación elegida asignado en la base de datos; se obtiene de "self.lista_items[0]" que es el primer elemento de la lista que se genera al seleccionar la fila (con **on_select()**) en la tabla de clientes. A través de la función se instancia el método **eliminar()** de la clase **Abmc()**, o sea **Abmc().eliminar()**, que intenta conectar a la base de datos (como se vio con el método **consultar()**) y en la tabla "registros" ejecuta la instrucción **"DELETE FROM registros WHERE registros.id=%s"** para eliminar el registro en la mencionada base de datos. Seguidamente con **vista_tasacion()** se actualiza la información en la tabla de la venta principal. Si también la tasación cuenta con imágenes se recorre la lista "self.ruta_imagenes" proveniente de **on_select()** para eliminar los archivos del directorio con **os.remove()**.

gestionar.py

```
def eliminar_registro(tabla, id):
    """
    Elimina un registro seleccionado en la tabla de la ventana principal del programa.
    """
    # Elimina el registro de la bd con el id correspondiente
    Abmc().eliminar(id)
    # Actualiza los datos de la tabla en la ventana principal del programa
    vista_tasacion(tabla)
```

abmc.py

```
def eliminar(self, id):
        """
        Se elimina el registro con el id indicado de la bd.
        """
        # Se intenta eliminar un registro
        try:
            cursor = self.conexion.cursor()
            # Se elimina el registro de la bd
            sql = "DELETE FROM registros WHERE registros.id=%s"
            cursor.execute(sql, (id,) )
            self.conexion.commit()
        except:
            messagebox.showwarning("Alerta", "No se pueden eliminar los datos")
        # Se cierra la conexión
        finally:
            if self.conexion:
                self.conexion.close()
```

4.6.3. Botón "Eliminar Todo"

Para eliminar todos los registros de la base de datos se utiliza el botón "Eliminar Todo" que acciona el método **eliminar_todo()**. El proceso es parecido al de "Eliminar" una fila con la excepción de que no es necesario pasar un identificador ("id") como argumento.

main.py

```
def eliminar_todo(self):
        """
        Al ejecutar el botón "Eliminar Todo" se eliminan los registros de la tabla en la bd.
        El método llama a la función "eliminar_todo" del módulo "gestionar.py" la cual se conecta a
        la bd a través de la clase "Abmc" y realiza la tarea requerida.
        """
        # Se verifica que se haya seleccionado una fila
        resultado =  messagebox.askquestion("Eliminar",
                                            "¿Está seguro que desea eliminar todos los registros?")
        if resultado == "yes":
            # Se eliminan todas las filas de la tabla en la base de datos
            # Se pasa como argumentos la tabla de tasaciones
            eliminar_todo(self.tabla)
            # Se eliminan todos los archivos en el directorio de trabajo (si hubiese)
            list(map( os.remove, (os.path.join(self.directorio_img,f)
                                            for f in os.listdir(self.directorio_img))))
```

Se instancia **Abmc().eliminar_todo()** dentro de la función **eliminar_todo()** (en el módulo “gestionar.py”) y se ejecuta la instrucción **“sql_del = "DELETE FROM registros"** que elimina todos los registros de la base de datos. También se establece que el id autoincremental reinicie en 1 con la instrucción **"ALTER TABLE registros AUTO_INCREMENT = 1".** Luego se actualiza la información en la tabla de clientes con **vista_tasacion()**, que en este caso la dejará vacía.

Con “list(map(os.remove, (os.path.join(self.directorio_img,f) for f in os.listdir(self.directorio_img))))” se listan todos los archivos existentes en el directorio de imágenes y se eliminan con **os.remove()**

gestionar.py

```
def eliminar_todo(tabla):
    """
    Elimina todos los registros existentes en la tabla de la ventana principal del programa.
    """
    # Elimina el registro de la bd con el id correspondiente
    Abmc().eliminar_todo()
    # Actualiza los datos de la tabla en la ventana principal del programa
    vista_tasacion(tabla)
```

abmc.py

```
def eliminar_todo(self):
        """
        Se eliminan todos las filas en la tabla "registros" de la bd.
        """
        # Se intenta eliminar un registro
        try:
            cursor = self.conexion.cursor()
            # Se elimina el registro de la bd
            sql_del = "DELETE FROM registros"
            # Se reinicia el id autoincremental en 1
            sql_inc = "ALTER TABLE registros AUTO_INCREMENT = 1"
            cursor.execute(sql_del)
            self.conexion.commit()
            cursor.execute(sql_inc)
            self.conexion.commit()
        except:
            messagebox.showwarning("Alerta", "No se pueden eliminar los datos")
        # Se cierra la conexión
        finally:
            if self.conexion:
                self.conexion.close()
```

4.6.4. Botones “Nuevo” y “Editar”

Ambos botones cuando se accionan (**nuevo_formulario()** y **editar_formuario()**) instancian la clase “Formulario” ubicada en el archivo “formulario.py” dentro de un widget Toplevel(), es decir, aparece una ventana emergente sobre la ventana principal. La diferencia que presentan ambas instancias se encuentra en los valores de los atributos del objeto que se crea.

main.py

```
def nuevo_formulario(self):
        """
        Al ajecutar el botón "Nuevo" se abre una ventana que contiene un formulario para cargar los
        datos de una nueva tasación.
        """
        popupFormulario = Toplevel()
        # Instanciación de la clase Formulario
        Formulario(popupFormulario, tabla = self.tabla,  directorio_img = self.directorio_img)
        # Restricciones en la ventana principal
        popupFormulario.grab_set()
        popupFormulario.focus_set()
        popupFormulario.wait_window()
        # Se eliminan los datos cargados previamente al seleccionar una tasación en la tabla
        self.limpiar_datos()

    def editar_formulario(self):
        """
        Al seleccionar una tasación en la tabla y al ejecutar el botón "Editar" se abre una ventana
        con un formulario que contiene los datos cargados previamente al crear dicha tasación.
        Los valores de la tasación seleccionada se pasan como argumento a la clase (como "entrys")
        y se utilizan para setear las entradas respectivas. También se pasan como argumento la tabla
        completa y las listas de imágenes (nombres de las mismas y ubicación en el directorio) para
        visualizar los archivos guardados en la bd y en el directorio en la tabla del formulario.
        """
        try:
            # Se verifica que se haya seleccionado una fila
            if self.seleccion_item_app:
                popupFormulario = Toplevel()
                # Instanciación de la clase Formulario
                Formulario(popupFormulario, tabla = self.tabla, entrys = self.lista_items,
                           ruta_file = self.ruta_imagenes, name_file = self.lista_imagenes,
                           directorio_img = self.directorio_img, accion = "editar")
                # Restricciones en la ventana principal
                popupFormulario.grab_set()
                popupFormulario.focus_set()
                popupFormulario.wait_window()
                # Se eliminan los datos cargados previamente al seleccionar una tasación en la tabla
                self.limpiar_datos()
                # Se elimina la selección vigente para obligar a seleccionar nuevamente una fila
                del self.seleccion_item_app
        except:
            messagebox.showinfo("Alerta", "Seleccione una fila")
```

Al crear un nuevo formulario solamente es necesario pasar como argumentos la ruta del directorio donde se guardarán las imágenes y el objeto de la tabla de clientes. Se abrirá por lo tanto una nueva ventana con entradas (o comboboxs) vacías para que el usuario complete y cargue imágenes.

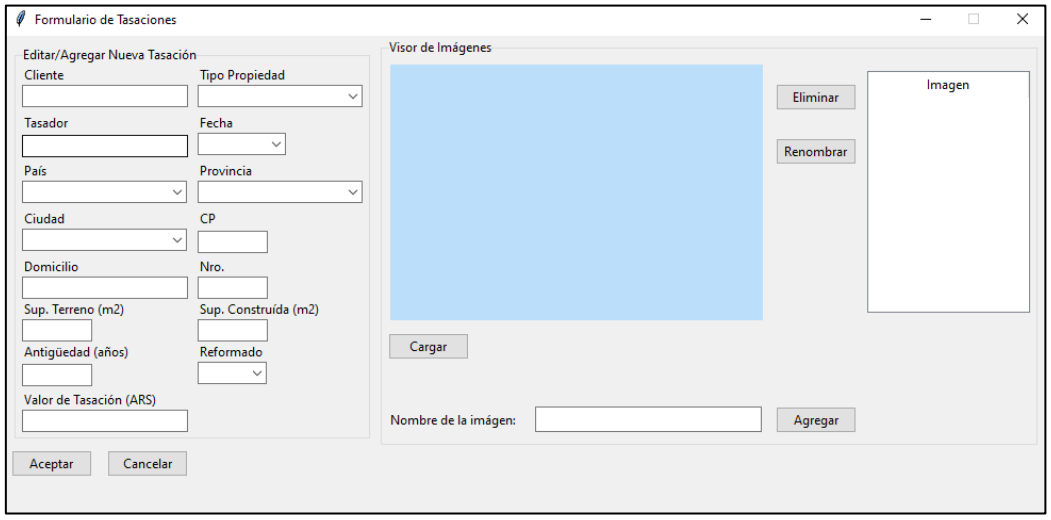

Figura 6 - Formulario nuevo

Por otro lado, al editar un formulario se deben incluir como parámetros del objeto las variables “self.lista_items” (valores de la tasación), “self.ruta_imagenes” (lista con la ruta de las imágenes de la tasación en el directorio) y self.lista_imagenes (lista con los nombres de las imágenes) provenientes de **on_select()**. Asimismo, se indica que la acción a realizar es de tipo “editar”.

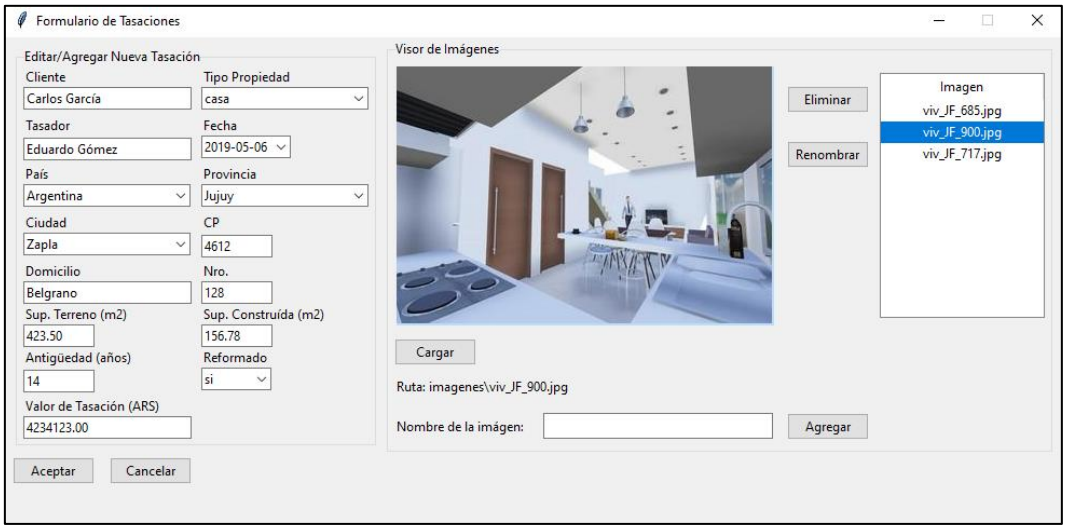

Figura 7 - Editar Formulario

Las entradas y comboboxs en el formulario están configuradas para mostrar valores por defecto. En el caso de crear un nuevo formulario los campos de los widgets quedan vacíos al tomarse el parámetro “entrys = [""]*17” preestablecido en la clase, pero al editar un formulario se insertan (usando **set()** o **insert()**) los valores de “self.lista_items”. Y en la tabla de imágenes se insertan los nombres de los archivos guardados a partir de “self.name_file”.

formulario.py

```
# Entradas
self.entry_cliente = ttk.Entry(self.marco_registros, textvariable = StringVar(), width = 25)
self.entry_cliente.delete(0, END)
self.entry_cliente.insert(0, self.entrys[1])
self.entry_cliente.grid(row = 2, column = 1, padx = (5,0), pady = (0, 5), sticky = W)
                                    foreground = 'white', borderwidth = 2, date_pattern ='y-mm-dd')
. . . . . .


# Comboboxs
self.cmbbox_tipo = ttk.Combobox(self.marco_registros, textvariable = StringVar(),
                                     width = 22)
self.cmbbox_tipo["values"] = ["Casa", "Departamento", "PH", "Terreno"]
self.cmbbox_tipo.set(self.entrys[2])
self.cmbbox_tipo.grid(row = 2 , column = 2, padx = (5,5), pady = (0,5), sticky = W)

. . . . . .

# ACCIONES EN LA TABLA DE IMÁGENES
# Se llena la tabla con las imágenes de la tasación existentes (si hubiese)
      for img in self.name_file:
          self.tabla_img.insert('', "end", text = img , values = img)
      # Cambiar imagen en el canvas al seleccionar en tabla
      self.tabla_img.bind('<<TreeviewSelect>>', self.on_select_img)
```

Dentro de los métodos **nuevo_formulario()** y **editar_formulario()**, al final, se ejecuta **limpiar_datos()**. Su objetivo es eliminar los datos en las entradas del área de información y eliminar la imagen del canvas en la ventana principal para que cuando se retorna de la ventana emergente (formulario) se manifieste que es necesario seleccionar nuevamente una tasación en la tabla de clientes para realizar alguna acción (editar, buscar o eliminar).

main.py

```
def limpiar_datos(self):
```

```
        """
        Se eliminan los datos cargados en las entradas del área de información de la tasación y la
        imagen existente en el canvas del área de imágenes
        """
        # Se elimina la imagen existente
        self.canvas_app.delete("all")
        # Se eliminan los datos en las entradas
        for v in range(len(self.entry_informacion)):
            self.entry_informacion[v].configure(state='normal')
            self.entry_informacion[v].delete(0, END)
            self.entry_informacion[v].configure(state='disable')
```

5. Módulo “formulario.py”

5.1. Interfaz gráfica de la ventana de formulario

Este módulo contiene la clase “Formulario” a través de la cual se desarrolla la ventana emergente del programa. Gráficamente se divide en tres secciones, una donde se ingresan los datos de las tasaciones (sección de registros), otra donde se ubican los botones principales (sección de botones principales) y la última es un área dedicada a la carga de imágenes (sección de imágenes).

- Sección de registros: se compone de objetos de entrada y combobox. Este es el lugar donde se ingresan las características de las tasaciones (Cliente, Tipo de propiedad, etc.)
- Sección de botones principales: comprende dos botones (“Aceptar” y “Cancelar”) para ratificar los cambios realizados en el formulario o descartarlos.
- Sección de imágenes: se compone de varios tipos de objetos dedicados a la carga y visualización de las imágenes asignadas a una tasación.

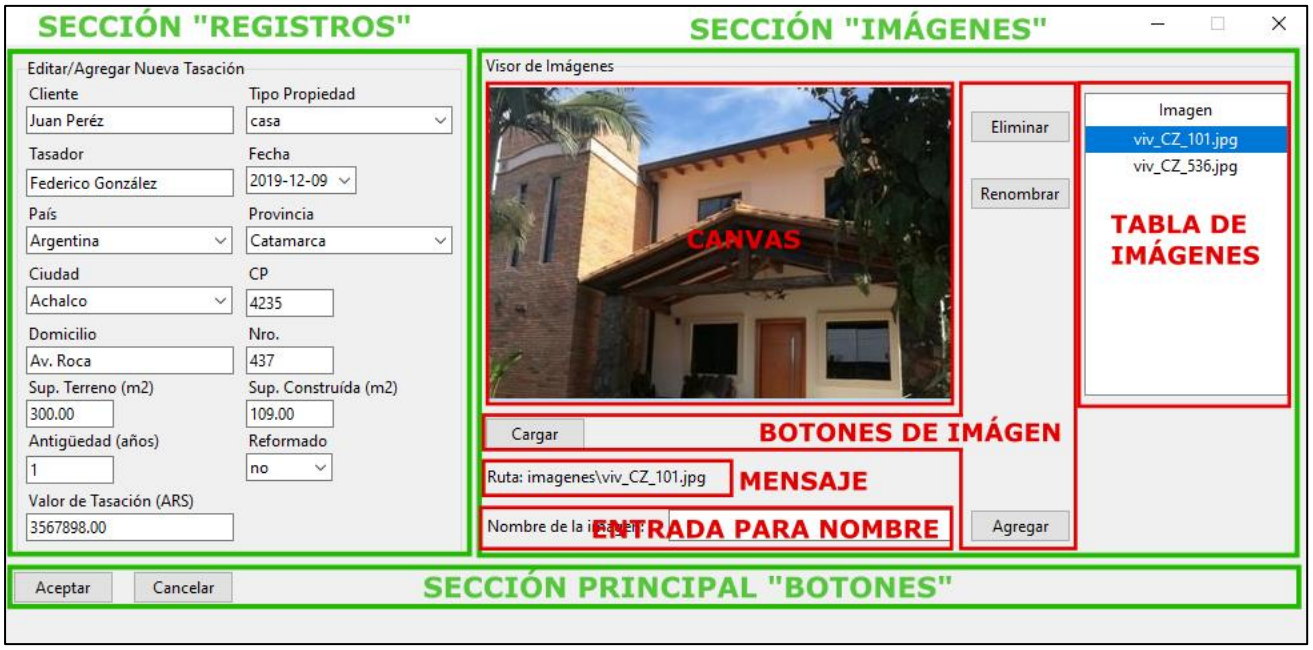

Figura 8 - Ventana de Formulario

formulario.py

```
def __create_widgets(self):
        """
        Se crean los widgets de la clase.
        """
        # CONTENEDOR PRINCIPAL
        # ----------------------------------------------------------------------------------------
        self.contenedor = Frame(self.parent)
        self.contenedor.pack(expand="yes", fill=BOTH)

        # SECCIONES PRINCIPALES
```

```
        # ------------------------------------------------------------------------------------------
        # Sección de ingreso de datos
        self.seccion_registros = ttk.Frame(self.contenedor)
        self.seccion_registros.grid(row = 0, column = 1)

        # Sección de imágenes
        self.seccion_imagenes = Frame(self.contenedor)
        self.seccion_imagenes.grid(row = 0, column = 2)

        # Sección de botones
        self.seccion_botones = Frame(self.contenedor)
        self.seccion_botones.grid(row = 1, column = 1, sticky = W)

        # ÁREA DE REGISTROS
        # ------------------------------------------------------------------------------------------
        # Contenedor del área
        self.area_registros = ttk.Frame(self.seccion_registros)
        self.area_registros.pack()
        . . . . . .

        # ÁREA DE IMÁGENES
        # ------------------------------------------------------------------------------------------
        # Contenedor del área
        self.area_imagenes = ttk.Frame(self.seccion_imagenes)
        self.area_imagenes.pack()

        # Labelframe
        self.marco_imagenes = LabelFrame(self.area_imagenes, text = "Visor de Imágenes")
        self.marco_imagenes.pack(expand="yes", fill=BOTH, pady = (0,0))
        . . . . . .

        # ACCIONES EN LA TABLA DE IMÁGENES
        for img in self.name_file:
            self.tabla_img.insert('', "end", text = img , values = img)
        # Cambiar imagen en el canvas al seleccionar en tabla
        self.tabla_img.bind('<<TreeviewSelect>>', self.on_select_img)

        # ÁREA BOTONES PRINCIPALES
        # ------------------------------------------------------------------------------------------
        # El boton "Aceptar" ejecuta diferentes métodos según la acción de formulario requerida
        if self.accion == "nuevo":
            self.btn_aceptar = Button(self.seccion_botones, text = '  Aceptar  ',
                                      command = self.datos_guardar)
            self.btn_aceptar.grid(row = 0, column = 1, sticky = W + E, padx = 7)
        elif self.accion == "editar":
            self.btn_aceptar = Button(self.seccion_botones, text = '  Aceptar  ',
                                      command = self.datos_modificar)
            self.btn_aceptar.grid(row = 0, column = 1, sticky = W + E, padx = 7)
        . . . . . .
```

5.2. Sección de registros

Se compone de widgets de entradas y combobox en los cuales se ingresan o eligen las características de una tasación. Cuando se crea un nuevo formulario los campos de estos widgets aparecen vacíos, mientras que si se edita un formulario existente se autocompletan con los valores cargados al crear el registro como se describió antes al explicar el funcionamiento del botón “Editar” de la ventana principal. Los campos dedicados a “Provincia”, “Ciudad” y “CP” utilizan métodos **get_localidad()** y **set_cp()** que son los mismos que se usan en el área buscar de la ventana principal. O sea, se selecciona una provincia de la lista desplegable, se filtran las ciudades en relación a la elección, y cuando se elige una localidad se inserta el código postal en la entrada designada.

5.3. Sección de imágenes

Esta sección contiene un canvas donde se proyectan las imágenes cargadas desde directorio. Al ejecutar el botón “Cargar” se acciona el método **cargar()** el cual inicialmente abre una ventana de archivos mediante el método **filedialog.askopenfilename()**. Este último lo que hace es capturar la ruta de la imagen cargada que se pasa luego por el método **vista_imagen()** (similar al utilizado en la clase “Aplicaciones”) que es el encargado

de abrir la imagen utilizando la librería Pillow e incrustarla en el canvas. Para este programa se restringe la carga de imágenes a la extensión “.jpg”.

formulario.py

```
 def cargar(self):
        """
        Se abre una ventana de archivos para cargar las imágenes. El código está preparado para
        trabajar solamente con archivos ".jpg". Al cargar la imágen ésta se visualiza en el canvas
        y se imprime debajo de él un mensaje con la ubicación del archivo.
        """
        # Se abre una ventana para cargar las imágenes
        self.ruta_cargar = filedialog.askopenfilename(title='Subir',
                                            filetypes = (("jpg files","*.jpg"),("all fi-
les","*.*")))
        # Se verifica que se haya seleccionado una imágen para cargar
        if len(self.ruta_cargar) < 1:
                self.canvas_form.delete("all")
                messagebox.showinfo("Alerta", "Debe seleccionar una imágen para cargar")
        else:
            # Se verifica que el formato de imágen sea el correcto
            if not self.ruta_cargar.lower().endswith(('.jpg')):
                messagebox.showwarning("Alerta",
                                       "Formato de imágen no válido. Cargue un archivo '.jpg'")
            else:
                # Se carga la imagen en el canvas
                self.vista_imagen(self.ruta_cargar, (350,235))
                # Se imprime la ruta de la imágen cargada
                self.ruta_imagen(self.ruta_cargar)
```

Debajo del canvas aparece un mensaje con la ruta de la imagen cargada, el mismo se genera con el método **ruta_imagen().** Simplemente limita la longitud del texto de la ruta a 58 caracteres y lo inserta en el objeto de mensaje “self.message_img”.

formulario.py

```
def ruta_imagen(self, ruta):
        """
        Imprime un mensaje con la ubicación del archivo, la cual se pasa como argumento. El mensaje
        se limita a 58 caracteres.
        """
        try:
            mensaje ="Ruta: " + str(ruta)
            if len(mensaje) > 58:
                mensaje = mensaje[0:58] + "..."
            self.message_img.config(text = mensaje)
        except:
            self.message_img.config(text = "...")
```

5.3.1. Botón “Agregar”

Si se considera guardar la imagen abierta desde el directorio, el siguiente paso es agregarla a la tabla de imágenes con el botón “Agregar”. Al presionarlo se ejecuta **agregar_entabla()** el cual guarda la ruta de la imagen cargada en una lista definida y en otra el nombre que el usuario le asigne. Posteriormente, el método **actualizar_tabla()** recorre la última lista mencionada para insertar los nombres de las imágenes definidos en la tabla de las imágenes.

En un formulario nuevo no hay imágenes cargadas, por lo tanto, la lista de nombres que utiliza la tabla de imágenes está vacía. El nombre de la variable que referencia a la lista es “self.name_file_op” , que es una copia de “self.name_file” (ambas variables se definen en el constructor de la clase). Al cargar una imagen, estando la tabla vacía (o la lista de nombres), “self.name_file_op” pasa a tomar el valor de “[self.ruta_guardar.get() + ".jpg"]”; en cambio si ya existe un nombre en la tabla entonces se adiciona el nuevo nombre, de una nueva imagen cargada, a lista con lista.**append()**.

De este modo, **actualizar_tabla()** recorre la lista con los nombres agregados y los inserta en la tabla con self.tabla_img**.insert()**, siendo “self.tabla_img” la variable que identifica dicha tabla.

Adicionalmente, se guarda la ruta de las imágenes cargadas en la lista “self.ruta_file_op” (copia de “self.ruta_file”).

Hasta este punto solamente hay dos listas, una con el nombre de las imágenes agregadas y otra con sus ubicaciones en el directorio. Las imágenes se guardarán en forma permanente solamente al apretar el botón “Aceptar”. Es por eso que se trabaja con “self.name_file_op” y “self.ruta_file_op” que son copias y se reiniciarán a los valores de “self.name_file” y “self.ruta_file” si se descartan los cambios cuando se cancela el formulario.

formulario.py

```
def agregar_entabla(self):
        """
        Cuando se carga una imagen desde el directorio se debe definir un nombre para ella en la
        entrada prevista. Al presionar el botón "Agregar" el nombre elegido se agrega a la lista
        "self.name_file_op" y se visualiza en la tabla. La ubicación del archivo se adiciona a la
        lista "self.ruta_file_op" en la misma posición que el nombre.
        """
        try:
            # Se verifica que la imágen que se quiere agregar o el nombre establecido no se repitan
            if self.ruta_cargar in self.ruta_file_op:
                messagebox.showinfo("Alerta", "El archivo ya fue cargado. Elija otra imágen")
            elif len(self.ruta_cargar) == 0:
                messagebox.showinfo("Alerta", "Cargue una imágen")
            elif (self.ruta_guardar.get() + ".jpg") in self.name_file_op:
                messagebox.showinfo("Alerta", "El nombre ya existe. Elija otro nombre")
            elif (self.ruta_guardar.get() + ".jpg") in os.listdir(self.directorio_img):
                messagebox.showinfo("Alerta", "El archivo ya existe. Elija otro nombre")
            else:
                # Se agrega la ruta de la imagen cargada y el nombre establecido a las listas
                # correspondientes
                if  len(self.ruta_guardar.get()) > 0:
                    # Cuando no hay imágenes guardadas en la tasación
                    if self.name_file_op == [""]:
                        self.ruta_file_op = [self.ruta_cargar]
                        self.name_file_op = [self.ruta_guardar.get() + ".jpg"]
                    # Cuando ya imágenes guardadas para la tasación
                    else:
                        self.ruta_file_op.append(self.ruta_cargar)
                        self.name_file_op.append(self.ruta_guardar.get() + ".jpg")
                else:
                    messagebox.showinfo("Alerta", "Ingrese un nombre válido", parent = self.parent)
                # Se actualiza la tabla
                self.actualizar_tabla()
        except:
            # Cuando se cancela la carga de un archivo de imagen
            messagebox.showinfo("Alerta", "Debe cargar una imágen del directorio")
```

5.3.2. Botón “Eliminar”

Así como se agregan imágenes también se pueden eliminar, tanto de “self.tabla_img” como del directorio (al apretar “Aceptar” como se explica más adelante). En el caso de la tabla, al seleccionar una imagen y ejecutar “Eliminar” se llama al método **eliminar_entabla()** que borra el nombre de la imagen en la tabla con self.tabla_img.**delete()** y en “self.name_file_op” a partir de su posición en la lista obtenida con **.index(***nombre de la imagen***)**. En la misma posición se elimina en “self.ruta_file_op” la ruta de la imagen en el directorio.

formulario.py

```
def eliminar_entabla(self):
        """
        Cuando se selecciona una imagen en la tabla y se ejecuta el botón "Eliminar", se elimina su
        nombre en dicha tabla y en la lista "self.name_file_op". En la misma posición de la lista
        de nombres se elimina la ruta en "self.ruta_file_op".
        """
        try:
            if self.seleccion_item:
                resultado =  messagebox.askquestion("Alerta",
                                                    "¿Está seguro que desea eliminar la imágen?")
                if resultado == "yes":
                    # Se borran items de la tabla
                    self.tabla_img.delete(self.seleccion_item)
                    # Se elimina la variable para que no exista si la tabla está vacía y evitar un error
```

```
                    del self.seleccion_item
                    # Se calcula la posición de la imagen en la lista "self.name_file"
                    indice = self.name_file_op.index(self.valor_item[0])
                    # Se elimina la imágen de las correspondientes listas
                    del self.name_file_op[indice]
                    del self.ruta_file_op[indice]
        except:
            messagebox.showinfo("Alerta", "Seleccione una imágen")
```

5.3.3. Botón “Renombrar”

Para renombrar una imagen se debe seleccionar un ítem en la tabla de imágenes, introducir un nombre válido en la entrada prevista y apretar el botón “Renombrar” el cual corre el método **renombrar_entabla()**. Lo que hace es capturar el nombre de la imagen en la tabla con self.tabla_img.**item()[“values”]**, buscar su posición en la lista “self.name_file_op” con **.index(***nombre de la imagen***)** y guardar en ese lugar el nombre nuevo. Dentro del método se ejecuta además **actualizar_tabla()** que actualiza los cambios en la tabla como ya se indicó.

formulario.py

```
def renombrar_entabla(self):
        """
        Al renombrar una imagen y confirmar se verifica que el nombre nuevo no exista. Los cambios
        se aplican temporalmente sobre la lista "self.name_file_op" y se visualizan en la tabla, pero
        si no se ratifican al apretar el botón "Aceptar" no persisten ya que no se actualizan la bd y
        el directorio donde se guardan los archivos.
        """
        try:
            if self.seleccion_item:
                if len(self.ruta_guardar.get()) == 0:
                    messagebox.showinfo("Alerta", "Ingrese un nombre válido")
                elif (self.ruta_guardar.get() + ".jpg") in os.listdir(self.directorio_img):
                    messagebox.showinfo("Alerta", "El archivo ya existe. Elija otro nombre")
                else:
                    resultado =  messagebox.askquestion("Alerta",
                                                        "¿Está seguro que desea renombrar la imagen?")
                    if resultado == "yes":
                        # valor (nombre) del ítem (imagen) seleccionado
                        self.valor_item = self.tabla_img.item(self.seleccion_item)["values"]
                        # Se obtiene el índice
                        indice = self.name_file_op.index(self.valor_item[0])
                        # Se renombra la imagen
                        if self.ruta_guardar.get() + ".jpg" in self.name_file_op:
                            messagebox.showinfo("Alerta", "El nombre ya existe. Elija otro nombre")
                        else:
                            # Se actualiza la lista "self.name_file_op" con el nuevo nombre ingresado
                            self.name_file_op[indice] = self.ruta_guardar.get() + ".jpg"
                            # Se actualiza la tabla con el nuevo nombre ingresado
                            self.actualizar_tabla()
                            # Se elimina la variable para que no exista si la tabla está vacía y evitar un e
rror
                            del self.seleccion_item
        except:
            messagebox.showinfo("Alerta", "Seleccione una imágen")
```

5.4. Sección de botones principales

5.4.1. Botón “Aceptar”

Nuevamente, se comenta que los botones “Agregar”, “Eliminar” y “Renombrar” no están realizando cambios en la base de datos o en el directorio de trabajo, solamente modifican los valores de las listas “self.name_file_op” y “self.ruta_file_op” que contienen el nombre de las imágenes y su ubicación en el directorio respectivamente. Para que los cambios persistan es necesario ratificar los cambios apretando el botón “Aceptar” que ejecuta **datos_guardar()** cuando se trata de un formulario nuevo o **datos_modificar()** si se edita un formulario existente.

Al crear un formulario nuevo y guardarlo lo primero que se hace es verificar que todos los campos en las entradas estén completos, si es así, se guardan en la base de datos y las imágenes cargadas (si hubiese) en el directorio local elegido (en este caso ya está definido como "…/imágenes", y si no existe se crea).

formulario.py

```
def datos_guardar(self):
        """
        Los cambios realizados al crear un nuevo formulario se ejecutan al apretar el botón
        "Aceptar". Los valores ingresados en las entradas se guardan en la bd junto con un string
        que contiene los nombres de las imágenes cargadas, las cuales se se guardan en el directorio
        establecido.
        """
        # Se capturan los datos ingresados en las entradas
        valores = self.get_entradas()
        # Se define el estado de los campos del formulario
        self.validar_campos(valores)
        # Se verifica que las entradas están llenas
        if self.validar_campos(valores):
            messagebox.showinfo("Alerta", "Complete todos los campos")
        else:
            consulta =  messagebox.askokcancel('Cerrar', "¿Desea guardar los cambios?")
            if consulta:
                # Se guardan los datos en la base de datos
                alta_tabla(self.tabla, valores)
                # Si no existe, se crea el directorio definido para guardar las imágenes subidas
                directorioParaImgs = os.path.join(self.directorio_img)
                if not os.path.exists(directorioParaImgs):
                    os.mkdir(directorioParaImgs)
                # Se guardan las imágenes cargadas (si hubiese) en el directorio
                for i in range(len(self.ruta_file_op)):
                    name_file = self.name_file_op[i]
                    with PILImage.open(self.ruta_file_op[i]) as abrir_img:
                        guardar_img = os.path.join(directorioParaImgs, name_file)
                        abrir_img.save(guardar_img)
                # Se cierra la ventana
                self.parent.destroy()
```

Los valores ingresados en las entradas del formulario se capturan en una lista con **get_entradas()** y con **validar_campos()** simplemente se verifica que no estén vacíos. En el programa no hay una restricción respecto al tipo de dato cargado (texto o numérico) ya que es una tarea que excede el alcance de este trabajo, pero para evitar errores se recomienda cargar correctamente los campos (por ejemplo, completar con números las entradas que se entiende son de tipo numéricas).

formulario.py

```
 def get_entradas(self):
        """
        Devuelve una lista con los valores de las entradas del formulario. Si es un formulario de
        edición se adiciona el valor del id de la tasación seleccionada.
        """
        # Se capturan los valores de las entradas
        valores = [self.entry_cliente.get(), self.cmbbox_tipo.get(), self.entry_tasador.get(),
                   self.entry_fecha.get(), self.cmbbox_pais.get(), self.cmbbox_provincia.get(),
                   self.cmbbox_ciudad.get(), self.entry_cp.get(), self.entry_domicilio.get(),
                   self.entry_nro.get(), self.entry_terreno.get(), self.entry_construido.get(),
                   self.entry_antiguedad.get(), self.cmbbox_reformado.get(), self.entry_valor.get(),
                   str(self.name_file_op)]
        # Se adiciona el id de la tasación si se edita un registro existente
        if self.accion == "editar":
            valores.append(self.entrys[0])
        return valores

 def validar_campos(self, valores):
        """
        Se recorre cada campo del formulario, si hay alguno vacío la variable "campos_vacios" se
        configura en "True".
        """
        campos_vacios = False
        # Se verifica que los campos estén llenos
        for v in valores[0:14]:
```

```
            if v == "":
                campos_vacios = True
        return campos_vacios
```

Si la validación de campos es correcta se llama en **datos_guardar()** a la función **alta_tabla()** del módulo "gestionar.py" que invoca al método **alta()** perteneciente a la clase "Abmc" (en "abmc.py"), es decir, se ejecuta **Abmc().alta()**. Este último intenta conectarse al servidor de MySQL como ya se explicó con el método **consulta()** para luego lanzar la instrucción "**INSERT INTO registros(cliente, tipo, …) VALUES (%s, %s** …)" e insertar una nueva fila en la base de datos con los datos ingresados por el usuario.

gestionar.py

```
def alta_tabla(tabla, args):
    """
    Crea un nuevo registro con los datos cargados en el formulario.
    """
    # Crea el registro en la bd
    Abmc().alta(args)
    # Actualiza los datos de la tabla en la ventana principal del programa
    vista_tasacion(tabla)
```

abmc.py

```
def alta(self, args):
        """
        Inserta un nuevo registro en la tabla "registros" de la bd "tasaciones".
        """
        # Se intenta insertar un registro en la bd
        try:
            cursor = self.conexion.cursor()
            # Se inserta nuevo registro en la bd
            sql = """
                INSERT INTO registros(cliente, tipo, tasador, fecha, pais, provincia, ciudad, cp,
                domicilio, nro, sup_util, sup_const, antiguedad, reformado, valuacion, imagenes)
                VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
                """
            cursor.execute(sql, args)
            self.conexion.commit()
        except:
            messagebox.showwarning("Alerta", "No se puede realizar la operación")
        # Se cierra la conexión
        finally:
            if self.conexion:
                self.conexion.close()
```

Si se agregan imágenes a la tasación en el formulario como ya se vio el nombre y la ruta de éstas se guardan en "self.name_file_op" y "self.ruta_file_op". Entonces para guardarlas en el directorio en forma permanente se recorren las listas y con la librería Pillow se van abriendo y guardando en la ruta "imágenes/nombredelaimagen". Finalmente se cierra la ventana emergente (Toplevel()) ya que se llama al método **destroy()** de tkinter.

Para que los cambios se observen en la tabla de la ventana principal luego de llamar a **Abmc().alta()** se acciona **vista_tasacion()** (ambos en "gestionar.py") quién se encarga de capturar los valores de la base de datos en insertarlos en la tabla de acuerdo a lo que se explicó anteriormente.

En el caso de editar un formulario existente, al apretar "Aceptar" en la ventana emergente se requieren realizar algunas operaciones distintas a **datos_guardar()**. Por eso se llama al método **datos_modificar()**.

formulario.py

```
  # ÁREA BOTONES PRINCIPALES
        # ------------------------------------------------------------------------------------------
        # El boton "Aceptar" ejecuta diferentes métodos según la acción de formulario requerida
        if self.accion == "nuevo":
            self.btn_aceptar = Button(self.seccion_botones, text = '  Aceptar  ',
                                      command = self.datos_guardar)
            self.btn_aceptar.grid(row = 0, column = 1, sticky = W + E, padx = 7)
        elif self.accion == "editar":
            self.btn_aceptar = Button(self.seccion_botones, text = '  Aceptar  ',
```

```
                                        command = self.datos_modificar)
            self.btn_aceptar.grid(row = 0, column = 1, sticky = W + E, padx = 7)
        self.btn_cancelar = Button(self.seccion_botones, text = '  Cancelar  ', command = self.cancelar)
        self.btn_cancelar.grid(row = 0, column = 2, sticky = W + E, padx=7)
```

La captura de los datos ingresados en el formulario y su validación es similar a lo planteado en **datos_guardar()**. Una sola diferencia es que se agrega a esos datos capturados el id en la base de datos de la tasación tomándolo de la lista "self.entrys" (se aplica "valores.append(self.entrys[0]" en **get_entradas()**).

formulario.py

```
def datos_modificar(self):
        """
        Los cambios que se realizan sobre el formulario, ya sea modificando los datos de la tasación
        en las entradas como las operaciones de carga, renombre y eliminación de imágenes persisten
        al apretar el botón "Aceptar" y confirmar la acción, donde se actualizan los campos en
        la bd y el directorio de imágenes.
        Cuando se renombra una imágen (y se confirman los cambios) se hace una copia de la misma en
        el directorio de trabajo con el nuevo nombre y luego se elimina el archivo copiado.
        """
        # Se capturan los datos ingresados en las entradas
        valores = self.get_entradas()
        # Se define el estado de los campos del formulario
        self.validar_campos(valores)
        # Se verifica que las entradas están llenas
        if self.validar_campos(valores):
            messagebox.showinfo("Alerta", "Complete todos los campos")
        else:
            consulta =  messagebox.askokcancel('Cerrar', "¿Desea guardar los cambios?")
            if consulta:
                # Se actualiza la base de datos con los valores editados
                update_tabla(self.tabla, valores)
                # Se guardan primero las imágenes cargadas en el directorio (si hubiese)
                # Ruta del directorio de imágenes
                directorioParaImgs = os.path.join(self.directorio_img)
                # Se guardan las imágenes si "self.name_file_op" no está vacío, o sea, hay imágenes
                if self.name_file_op != [""]:
                    for i in range(len(self.ruta_file_op)):
                        name_file = self.name_file_op[i]
                        with PILImage.open(self.ruta_file_op[i]) as abrir_img:
                            guardar_img = os.path.join(directorioParaImgs, name_file)
                            abrir_img.save(guardar_img)
                # Se obtienen los nombres de las imágenes modificadas
                name_modificado = list(set(self.name_file) - set(self.name_file_op))
                # Si hay imágenes renombradas o eliminadas se crea una lista con su ubicación,
                # sino queda vacía
                if len(name_modificado) != [""]:
                    imagenes_modificadas = [self.directorio_img + i for i in name_modificado]
                else:
                    imagenes_modificadas = []
                # Se eliminan imágenes del directorio (las que fueron eliminadas o renombradas de la
                # tabla, si hubiese)
                for imagen in sorted(imagenes_modificadas, reverse = True):
                # Se hace esta verificación para evitar errores cuando la lista está vacía
                    if imagen != self.directorio_img:
                        os.remove(imagen)
                # Se cierra la ventana
                self.parent.destroy()
```

El id se utiliza para saber que fila está siendo editada en la base de datos. La operación de actualización se lleva adelante con la función **update_tabla()** del módulo gestionar.py quién llama al método **update()** de la clase "Abmc" (en abmc.py) para conectarse a la tabla "registros" de la base de datos y ejecutar **"UPDATE registros SET cliente=%s … WHERE registros.id=%s"**. Luego se llama a **vista_tasación()** para que al cerrarse la ventana emergente los datos en la ventana principal queden actualizados.

gestionar.py

```
def update_tabla(tabla, args):
```

```
    """
    Actualiza un registro con las modificaciones realizadas en el formulario.
    """
    # Actualiza el registro modificado en la bd
    Abmc().update(args)
    # Actualiza los datos de la tabla en la ventana principal del programa
    vista_tasacion(tabla)
```

abmc.py

```
def update(self, args):
        """
        Actualiza el registro designado en la tabla "registros" de la bd "tasaciones".
        """
        # Se intenta actualizar un registro
        try:
            cursor = self.conexion.cursor()
            # Se actualiza el registro designado en la bd
            sql = """
                UPDATE registros SET cliente=%s , tipo=%s, tasador=%s, fecha=%s, pais=%s,
                provincia=%s, ciudad=%s, cp=%s, domicilio=%s, nro=%s, sup_util=%s, sup_const=%s,
                antiguedad=%s, reformado=%s, valuacion=%s, imagenes=%s WHERE registros.id=%s
                """
            cursor.execute(sql, args)
            self.conexion.commit()
        except:
            messagebox.showwarning("Alerta", "No se pueden actualizar los datos")
        # Se cierra la conexión
        finally:
            if self.conexion:
                self.conexion.close()
```

Si hay imágenes en la tasación se guardan en el directorio con los nombres establecidos en “self.name_file_op” dada su ubicación en el directorio local en “self.ruta_file_op”. En este momento pueden pasar dos cosas. Si al editar el formulario no se modificó el nombre de una imagen (o se eliminó sacándola de las listas “self.name_file_op” y “self.ruta_file_op”) se va a volver a guardar encima de la imagen ya existente (con el mismo nombre y ubicación) en el directorio, es decir, se sobrescribe el archivo. En el caso de que se renombrara una imagen se va a guardar en el directorio con el nuevo nombre en la ruta “imagenes/nuevonombre” siendo “imagenes/” el directorio definido en el programa, dicho de otro modo, se hace una “copia” de la imagen renombrada y se guarda con otro nombre.

El siguiente paso será entonces eliminar en el pc las imágenes originales renombradas, ubicadas en “imagenes/viejonombre”, y las que fueron eliminadas de la tabla de imágenes y las listas correspondientes. Para ello, se calcula “name_modificado = list(set(self.name_file) - set(self.name_file_op))” donde se obtienen los nombres de las imágenes que sufrieron algún tipo de cambio (se eliminaron o renombraron). Ya que se guardan en el directorio “imagenes/” (“self.directorio_img”) se genera una lista de rutas con “imagenes_modificadas = [self.directorio_img + i for i in name_modificado]” la cual finalmente se recorre para concretar la eliminación de los archivos modificados (si hubiese) con **os.remove(imagen)**. Poy último, se cierra la ventana de edición del formulario, aplicando **destroy()**, y se vuelva a la ventana principal del programa.

5.4.2. Botón “Cancelar”

Si en lugar de aceptar los cambios se descartan al presionar “Cancelar”, directamente se cierra la ventana y se vuelve a la ventana principal sin hacer modificaciones en la base de datos o el directorio local. Al seleccionar otra tasación o intentar volver a editar la misma luego de cancelar los cambios, “self.name_file_op” y “self.ruta_file_op” copiaran los valores de “self.name_file” y “self.ruta_file” los cuales nunca varían salvo que si se acepten los cambios luego de editar un formulario.

formulario.py

```
def cancelar(self):
        """
        Destruye la ventana del formulario y vuelve a la ventana principal.
        """
        consulta =  messagebox.askokcancel('Cerrar', "¿Desea abandonar la aplicación?")
```

```
        if consulta:
            self.parent.destroy()
```