PRÁCTICA 2
CENTRO VETERINARIO: UNMASCOTA

ASIGNATURA
Programación orientada a objetos

GRUPO
Grupo 2

EQUIPO
Equipo 6

PROFESOR
Jaime Alberto Guzmán Luna

INTEGRANTES

López González Alejandro

Betancur Uribe Emmanuel

Martínez Ríos Santiago

Bula Fuente Melanie

Ospina Gaviria Tomas

Universidad Nacional de Colombia

Sede Medellín

2025

Índice

1. Portada

Nombre activi▪dad: Práctica 2

- **Nombre proyecto:** Centro Veterinario: UNamascota
- **Materia:** Programación Orientada a Objetos
- **Grupo:** 02
- **Equipo:** 6
- **Integrantes:** López González Alejandro, Betancur Uribe Emmanuel, Martínez Ríos Santiago, Bula Fuente, Melanie, Ospina Gaviria Tomas
- **Profesor:** Jaime Alberto Guzmán Luna
- **Nombre de la universidad:** Universidad Nacional de Colombia - Sede Medellín
- **Fecha:** 2025

2. Descripción general de la solución.

El análisis del dominio del proyecto es el primer paso en el desarrollo de la aplicación. En esta fase, se identifican y definen los requisitos y necesidades del proyecto. Este análisis incluye:

Centro Veterinario: UNamascota

UNamascota es un proyecto innovador que busca promover el cuidado responsable de animales domésticos y mejorar el bienestar animal en Medellín y sus alrededores. Con sedes estratégicamente ubicadas, UNamascota ofrece una plataforma integral para facilitar el proceso de cuidado animal y brindar servicios complementarios para el cuidado de las mascotas.

Objetivos Clave

- Facilitar el proceso de cuidado veterinario, asegurando que los animales estén en óptimas condiciones de salud y brindando una correcta retroalimentación sobre el estado de las mascotas.
- Promover la tenencia responsable de mascotas a través de educación y acompañamiento.
- Ofrecer una variedad de servicios para el cuidado integral de las mascotas, desde atención veterinaria hasta guardería y peluquería.
- Proveer productos de calidad para el bienestar de los animales en la tienda del proyecto.
- Ofrecer servicios de cremación y entierro digno para mascotas fallecidas, brindando un espacio de recuerdo y despedida.

Servicios Ofrecidos

1. **Emergencia Veterinaria:** Los usuarios pueden visitar cualquiera de las tres sedes de UNamascota para conocer el estado de sus mascotas y determinar si requieren hospitalización.
2. **Agendamiento de Servicios:** Los usuarios pueden reservar citas con especialistas en diferentes áreas a través de la plataforma de UNamascota.
3. **Funeraria:** En caso de fallecimiento de una mascota, UNamascota ofrece servicios de cremación y entierro, así como un espacio para que los usuarios puedan visitar y dejar flores en honor a su compañero.
4. **Tienda:** UNamascota cuenta con una tienda donde los usuarios pueden adquirir productos de calidad para el cuidado y bienestar de sus mascotas.
5. **Dieta Personalizada:** Muchos animales tienen necesidades específicas según sus características; por ello, planificar una dieta adecuada y conocer el estado nutricional de tu mascota es de suma importancia.

Objetivo del Centro Veterinario: UNamascota

UNamascota está diseñado para abordar y resolver diversas necesidades en el ámbito del cuidado responsable de mascotas y el bienestar animal. Su implementación tiene como finalidad facilitar la interacción entre los dueños y los animales, así como optimizar los servicios ofrecidos por el centro veterinario.

Necesidades a Solucionar

Optimización del Proceso de Emergencia Veterinaria:

- **Problema:** Los dueños suelen tener dificultades para identificar enfermedades en sus mascotas.
- **Solución:** El software proporcionará una interfaz amigable para que los usuarios puedan reportar los síntomas de sus mascotas. El sistema calculará la gravedad de la situación y determinará si es necesario hospitalizar al animal.

Centralización de Servicios para el Bienestar de las Mascotas:

- **Problema:** Los dueños necesitan acceder a múltiples servicios de manera eficiente para el cuidado de sus mascotas.
- **Solución:** El sistema permitirá agendar citas y acceder a servicios complementarios desde una única plataforma, mejorando la experiencia y comodidad del usuario.

Gestión de Servicios Funerarios:

- **Problema:** La pérdida de una mascota es un momento difícil, y los dueños requieren opciones seguras y de calidad para despedirse dignamente.
- **Solución:** El software ofrecerá información sobre servicios funerarios, incluyendo cremación y entierro, y permitirá a los usuarios gestionar estos servicios de manera sencilla.

Integración de Tienda en Línea:

- **Problema:** Los dueños suelen enfrentarse a catálogos extensos de productos, sin saber cuáles son los más adecuados para sus mascotas.
- **Solución:** Los usuarios podrán adquirir productos esenciales para sus mascotas utilizando filtros que faciliten encontrar alimentos, juguetes, accesorios e insumos para el hogar, todo centralizado en la misma plataforma que gestiona los servicios veterinarios y complementarios.

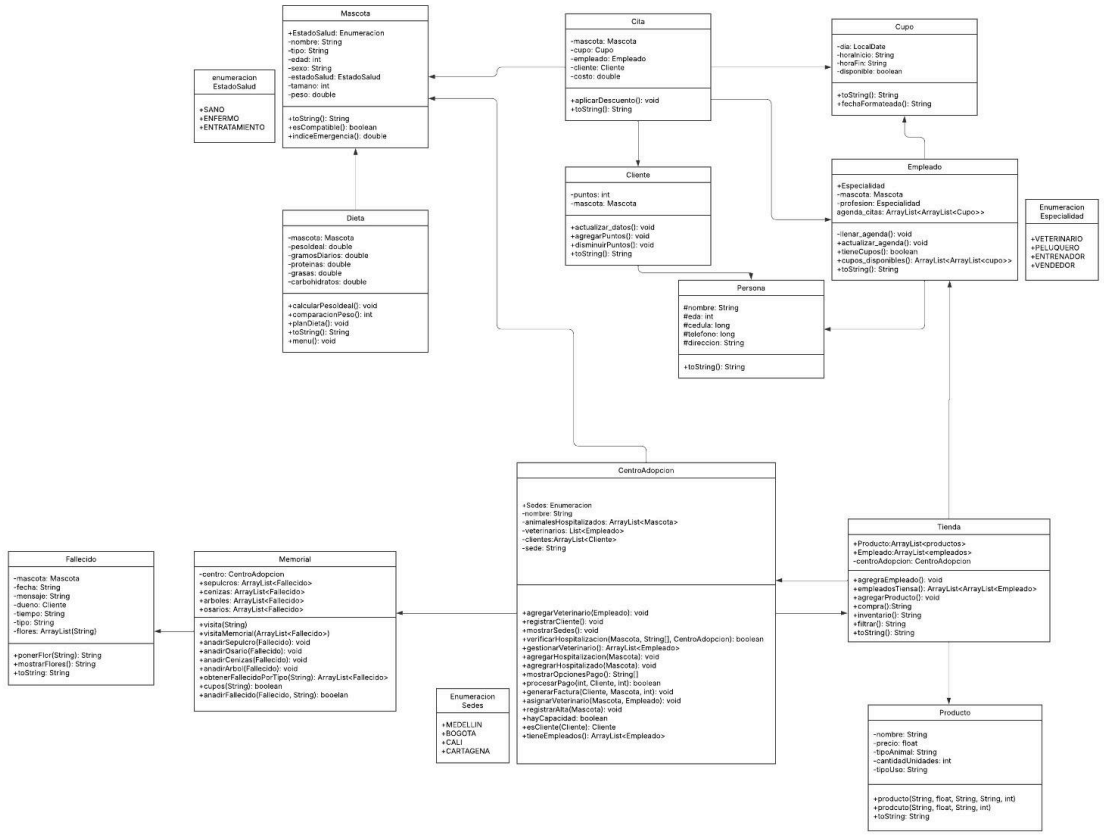
Sistema de Planificación de Dietas:

- **Problema:** Los dueños de mascotas suelen descuidar el aspecto de la salud nutricional de sus mascotas.
- **Solución:** Desde UNamascota se puede hacer un análisis de las condiciones nutricionales de su mascota para comprender la solución pertinente que se debe implementar respecto a su alimentación.

3. Descripción del diseño estático del sistema en la especificación UML.

En esta sección, se debe incluir el diagrama de clases que representa el dominio de la aplicación. Este diagrama debe mostrar todas las clases principales del sistema junto con las relaciones de cardinalidad entre ellas. La cardinalidad indica cuántas instancias de una clase pueden estar asociadas con una instancia de otra clase, proporcionando una visión clara de las interacciones y dependencias dentro del sistema.

Además del diagrama de clases, es importante proporcionar una breve descripción de la función de cada una de las clases en el proyecto. Esta descripción debe incluir:



1. CentroAdopcion

Propósito:
Hace referencia a las sedes, gestiona la operación del centro de adopción, incluyendo clientes, empleados, animales, citas y adopciones. Facilita el proceso de adopción y la prestación de servicios relacionados.

Relaciones:

- Cero a muchos con **Empleado**.
- Cero a muchos con **Mascota**.
- Cero a muchos con **Cliente**.

- Cero a muchos con **Cita**.
- Cero a uno con **Tienda**.
- Cero a uno con **Memorial**.

2. Mascota

Propósito:
Representa a los animales disponibles para adopción. Incluye atributos como nombre, tipo, edad, sexo y estado de salud. También representa a las mascotas que un usuario registre para recibir un servicio o agendar una cita.

3. Persona

Propósito:
Representa a las personas relacionadas con el centro de adopción, ya sean clientes o empleados. Almacena información personal básica.

Relaciones:

- Es una superclase abstracta de la que heredan **Cliente** y **Empleado**.

4. Empleado

Propósito:
Representa al personal del centro, que puede ser veterinario, peluquero, cuidador o tendero. Gestiona la atención a los clientes y animales.

Relaciones:

- Uno a muchos con **Cupo**.

5. Cupo

Propósito:
Representa un bloque de tiempo disponible para citas asignado a cada empleado. Incluye información sobre el día, la hora de inicio y fin, y si está disponible.

Relaciones:

- Uno a uno con **Cita** (cada cita utiliza un único cupo).

6. Cliente

Propósito:
Representa a los usuarios que buscan adoptar animales y utilizar otros servicios del centro, como la tienda y citas.

Relaciones:

- Uno a muchos con Mascota (Un Cliente puede tener múltiples mascotas).
- Uno a muchos con **Cita** (un cliente puede agendar múltiples citas).

7. Cita

Propósito:
Almacena información sobre las citas agendadas para servicios como veterinaria, guardería o peluquería. Incluye detalles del animal, cliente, empleado y costo.

Relaciones:

- Uno a uno con Mascota, **Cliente** y **Empleado** (cada cita está asociada a un único animal, cliente y empleado).
- Uno a uno con **Cupo** (cada cita ocupa un cupo específico).

8. Producto

Propósito:
Representa los artículos en venta en la tienda, incluyendo nombre, precio, tipo de animal y cantidad disponible.

Relaciones:

- Uno a muchos con **Tienda** (una tienda puede tener múltiples productos).

9. Tienda

Propósito:
Gestiona el inventario de productos disponibles para la venta, permitiendo a los clientes realizar compras.

Relaciones:

- Uno a muchos con **Producto** (una tienda puede tener múltiples productos).
- Uno a muchos con **Empleado** (los empleados pueden gestionar la tienda).

10. Memorial

Propósito:
Proporciona servicios de cremación y entierro para mascotas fallecidas, gestionando la memoria de los animales.

Relaciones:

- Uno a muchos con **Fallecido** (puede haber múltiples registros de mascotas fallecidas).

11. Fallecido

Propósito:
Representa a los animales que han fallecido, almacenando información sobre el dueño, fecha de fallecimiento y mensajes de recuerdo.

Relaciones:

- Uno a uno con **Mascota**(cada registro de "Fallecido" se refiere a un único animal fallecido).

- Uno a uno con **Cliente** (cada registro está asociado a un único dueño).

12. Dieta

Propósito:
Representa los requerimientos e información nutricional de cada mascota usado para recetar diferentes alimentaciones especiales.

Relaciones:

- Uno a uno con **Mascota** (Una mascota tiene una única dieta)

4. Descripción de la implementación de características de programación orientada a objetos en el proyecto.

En el enunciado de la práctica se solicitó la implementación de varias características de programación orientada a objetos.

- Interfaz Inicial:

- Segunda Interfaz:

- Field Frame:

```python
from tkinter import *
from tkinter import ttk
class FieldFrame(Frame):
    def __init__(self, ventana, tituloCriterios="", criterios=[], tituloValores="", valores=None, habilitado=None):
        super().__init__(ventana)
        self._tituloCriterios = tituloCriterios
        self._criterios = criterios
        self._tituloValores = tituloValores
        self._valores = valores
        self._habilitado = habilitado
        self._elementos = []

        # Configuración de estilo
        self.configure(bg='#f7f7f7')  # Fondo más claro
        estilo_etiqueta = {'font': ("Poppins", 12, 'bold'), 'bg': '#f7f7f7', 'fg': '#333'}
        estilo_campo = {'font': ("Poppins", 11), 'bg': '#ffffff', 'fg': '#333', 'bd': 2, 'relief': GROOVE}

        # Crear y colocar titulo de los criterios
        etiquetaTituloCriterios = Label(self, text=tituloCriterios, font=("Poppins", 14, 'bold'), bg='#f7f7f7', fg='#000
        etiquetaTituloCriterios.grid(column=1, row=0, padx=(10, 10), pady=(10, 10), sticky='ew')

        # Crear y colocar titulo de los valores
        etiquetaTituloValores = Label(self, text=tituloValores, font=("Poppins", 14, 'bold'), bg='#f7f7f7', fg='#000')
        etiquetaTituloValores.grid(column=2, row=0, padx=(10, 10), pady=(10, 10), sticky='ew')

        # Crear y colocar las etiquetas y campos de entrada para cada criterio
        for i in range(len(criterios)):
            etiquetaCriterio = Label(self, text=criterios[i], **estilo_etiqueta)
            etiquetaCriterio.grid(column=1, row=i + 1, padx=(10, 10), pady=(5, 5), sticky='e')

            campoValor = Entry(self, **estilo_campo)
            campoValor.grid(column=2, row=i + 1, padx=(10, 10), pady=(5, 5), sticky='ew')
```

gestorAplicacion.elementos.Persona (Línea 11)

```python
from abc import ABC

class Persona(ABC):
    def __init__(self, nombre, edad, cedula, telefono= None, direccion=None):
        self.nombre = nombre
        self.edad = edad
        self.cedula = cedula
        self.telefono = telefono
        self.direccion = direccion
```

Se decidió utilizar la clase Persona como clase abstracta, debido a que como persona no interviene directamente con la creación de objetos y cumple más un papel de guía para las subclases que heredan de ella y asÍ hay una mejor optimización de recursos.

- Herencia:

gestorAplicacion.elementos.Cliente (Línea 17)

```python
class Cliente(Persona):
    EDAD_MINIMA = 18

    def __init__(self, nombre, edad, cedula, telefono=None, direccion=None):
        super().__init__(nombre, edad, cedula, telefono, direccion)
        self.puntos = 0
        self.mascota = None
```

Se decidió hacer uso de la herencia en las clases Cliente y Persona, de manera que Persona herede a Cliente, para reutilizar atributos que tienen tanto la clase como la Cliente, y así optimizar mejor nuestras líneas de código.

- Ligadura Dinámica:

  gestionAplicacion.elementos.Persona (Linea 41)

```python
    def __str__(self):
        return (f"Nombre: {self.nombre}, Edad: {self.edad}, Cedula: {self.cedula}, "
                f"Telefono: {self.telefono}, Direccion: {self.direccion}")
```

Si hay clases que heredan de Persona y sobrescriben métodos como toString(), entonces se aprovechará la ligadura dinámica para resolver esos métodos en tiempo de ejecución.

- Atributos de Clase Y Métodos de Clase:

  gestionAplicacion.gestion.Tienda (Linea 25)

  gestionAplicacion.gestion.Tienda (Linea 26)

  gestionAplicacion.gestion.Tienda (Linea 51)

  gestionAplicacion.gestion.Tienda (Linea 55)

```
21      @staticmethod
22      def get_productos():
23          return Tienda.productos
24
25      @staticmethod
26      def get_empleados():
27          return Tienda.empleados
```

```
5   class Tienda:
6       # Listas estáticas para la serialización
7       productos = []
8       empleados = []
```

Se definen las listas como estáticas porque es necesario que estén para todos los objetos creados, y poder hacer llamadas desde la clase como el que se aplican en el deserealizador.

- Constante:

  gestorAplicacion.elementos.cliente (Linea 16)

```
15  class Cliente(Persona):
16      EDAD_MINIMA = 18
```

Se aplica la EDAD_MINIMA como constante

- Encapsulamiento:

  gestionAplicacion.elementos.Persona (Línea 14 - 18)

```
11  public abstract class Persona implements Serializable {
12      private static final long serialVersionUID = 1L;
13      //---> Atributos <---
14      protected String nombre;
15      protected int edad;
16      protected long cedula;
17      protected long telefono;
18      protected String direccion;
```

- Manejo self:

```
7   class Dieta:
8       def __init__(self, mascota):
9           self.mascota = mascota
10          self.pesoIdeal = 0
11          self.gramosDiarios = 0
12          self.proteinas = 0
13          self.grasas = 0
14          self.carbohidratos = 0
```

```
86      def esCompatible(self, otraMascota):
87          if self.getTipo() == otraMascota.getTipo() and self.getEstadoSalud() == otraMascota.getE
88              return True
89          else:
90              return False
```

- Enumeración:

  gestionAplicacion.elementos.CentroAdopcion (Línea 5)

```
5   class Sedes(Enum):
6       MEDELLIN = 1
7       BOGOTA = 2
8       CALI = 3
9       CARTAGENA = 4
```

  gestionAplicacion.elementos.estadoSalud

```
1   from enum import Enum
2   class EstadoSalud(Enum):
3       SANO = "SANO"
4       ENFERMO = "ENFERMO"
5       ENTRATAMIENTO = "EN TRATAMIENTO"
```

5. Descripción de cada una de las 5 funcionalidades implementadas.

- **Descripción de la funcionalidad:** Proporcione una descripción detallada y completa del comportamiento de cada funcionalidad. Asegúrese de explicar claramente cada una de las interacciones que componen la funcionalidad, describiendo cómo los diferentes componentes del sistema colaboran para lograr el objetivo de la funcionalidad.
- **Diagrama de interacción:** Incluya una imagen del diagrama de interacción correspondiente a cada funcionalidad. Asegúrese de que el diagrama sea claro y permita observar el comportamiento general de la funcionalidad, destacando las interacciones clave y los flujos de datos entre los componentes del sistema.

❖ FUNCIONALIDAD 1: Emergencia Veterinaria

La funcionalidad Emergencia Veterinaria comienza llamando el método registro() de la clase Cliente, el cual le pide al usuario un conjunto de datos como nombre, edad y cédula, para así crear un objeto de tipo Cliente que servirá como un identificador para el usuario. De la misma forma, el usuario ingresará las propiedades de su mascota como nombre, especie, edad, sexo, tamaño y peso, además de los síntomas que presenta la misma, así, se instanciará un objeto de tipo Mascota con dichos parámetros, el cual tendrá su atributo "estadoSalud" asignado como "enfermo" por defecto, reflejando la urgencia médica.

```
Cliente cliente = Cliente.registro();
```

```
public static Cliente registro(){
    ArrayList<Object> datos = Main.capturarDatosCliente();
    Cliente cliente = new Cliente((String) datos.get(0), (int) datos.get(1), (long) datos.get(2));
    cliente.agregarPuntos(50000);

    return cliente;

}
```

```
public static ArrayList<Object> capturarDatosCliente() {
    try {Scanner sc = new Scanner(System.in);
    System.out.println("---------------------");
    System.out.println("\n- Ingrese sus datos");
    System.out.print("\n- Nombre Completo: ");
    String nombreC = sc.nextLine();
    System.out.print("- Edad: ");
    int edadC = sc.nextInt();
    sc.nextLine();
    System.out.print("- Cédula: ");
    long cedula = sc.nextLong();
    sc.nextLine();
    System.out.println("\n---------------------");

    ArrayList<Object> datos = new ArrayList<>();
    datos.add(nombreC);
    datos.add(edadC);
    datos.add(cedula);

    return datos;

    } finally {}

}
```

```
Mascota mascota1 = new Mascota(nombreMascota, tipo, edadMascota, sexo, EstadoSalud.ENFERMO, tamano, peso);
```

A continuación, el sistema muestra las sedes disponibles mediante el método mostrarSedes() de la clase CentroAdopcion, el cual accede a su lista de sedes y las muestra al usuario. Una vez seleccionada la sede, el sistema valida su disponibilidad para atender la emergencia, esto se realiza con el método verificarHospitalizacion() de CentroAdopcion que:

- Comprueba si hay capacidad disponible en la sede, verificando que la cantidad de mascotas hospitalizadas sea menor a 10.

```
public boolean hayCapacidad() {
    if (animalesHospitalizados.size() >= 10) {
        return false;
    }
    return true;
}
```

- Utiliza el método gestionarVeterinario() de la clase CentroAdopcion para verificar que la sede tenga veterinarios disponibles, haciendo uso de tieneCupos de la clase Empleado.

```
public List<Empleado> gestionarVeterinario() {
    List<Empleado> disponibles = new ArrayList<>();
    for (Empleado veterinario : veterinarios) {
        if (veterinario.tieneCupos()) {
            disponibles.add(veterinario);
        }
    }
    return disponibles.isEmpty() ? null : disponibles;
}
```

- Utiliza el método esCompatible() de la clase Mascota para verificar que el nuevo animal no represente un riesgo para los animales ya hospitalizados, basándose en parámetros como tipo y estado de salud.

```
public boolean esCompatible(Mascota mascota) {
    if (this.tipo != mascota.tipo && this.estadoSalud != mascota.estadoSalud) {
        return false;
    }
    return true;
}
```

- Evalúa tres factores del animal para determinar si requiere hospitalización inmediata:
    - Gravedad de los síntomas (de 1 a 10).
    - Edad del animal.
    - Indicador de compatibilidad con otros animales (de 0 a 10).

  El sistema calcula un Índice de Emergencia (IE) con la siguiente fórmula:

  $$IE = (\text{Gravedad} \times 0.7) + (\text{Vulnerabilidad} \times 0.3) - (\text{Compatibilidad} \times 0.1)$$

```java
public double indiceEmergencia(int gravedad, int compatibilidad) {
    double vulnerabilidad = 10/(1+Math.abs(edad-4));
    double ie = (gravedad*0.7) + (vulnerabilidad*0.3) + (compatibilidad*0.1);

    return ie;
}
```

Donde:

- Gravedad es un valor calculado a partir de los síntomas.
- Vulnerabilidad es una función de la edad del animal:
    - Animales jóvenes (<1 año) y ancianos (>8 años) tienen valores altos de vulnerabilidad.
    - Vulnerabilidad = 10/(1+|Edad-4|) (edad más cercana a 4 es menos vulnerable).
- Compatibilidad es un valor calculado a partir de los síntomas, mide el riesgo de interacción con otros animales (valor bajo indica menor riesgo).

Si el índice de emergencia (IE) supera el umbral con valor predeterminado de 7.0, el animal se hospitaliza.

```java
public boolean verificarHospitalizacion(Mascota mascota, String[] listaSintomas, CentroAdopcion centro) {

    int gravedad = 0;
    int compatibilidad = 0;

    //Asignar valores de gravedad y compatibilidad a los síntomas
    for (String sintoma : listaSintomas) {
        switch (sintoma.toLowerCase()) {
            case "fiebre":
                gravedad += 2;
                compatibilidad += 3;
                break;
            case "vomito":
            case "vómito":
                gravedad += 3;
                compatibilidad += 2;
                break;
            case "picazon":
            case "picazón":
                gravedad += 2;
                compatibilidad += 1;
                break;
            case "enrojecimiento":
                gravedad += 1;
                compatibilidad += 2;
                break;
            case "inflamacion":
            case "inflamación":
                gravedad += 2;
                compatibilidad += 2;
                break;
            case "y":
                break;
            default:
                //System.out.println("\nSíntoma desconocido: " + sintoma);
                break;
        }
    }

    if (mascota.indiceEmergencia(gravedad, compatibilidad) < 7.0) {
        return false;
    }
```

```
    if (centro.gestionarVeterinario().isEmpty()) {
        return false;
    }

    if (!hayCapacidad()) {
        return false;
    }

    for (Mascota hospitalizado : animalesHospitalizados) {
        if (!mascota.esCompatible(hospitalizado)) {
            return false;
        }
    }
    return true;
}
```

Cuando se confirma si la mascota requiere hospitalización, se procede a listar los veterinarios disponibles. Esto se realiza mediante el método gestionarVeterinario() de la clase CentroAdopcion, que itera sobre la lista de veterinarios de la sede y verifica su disponibilidad con el método tieneCupo() de la clase Empleado, y el método asignarVeterinario() de la clase CentroAdopcion, que asigna el veterinario que elige el usuario al atributo "veterinario" del objeto de tipo Mascota instanciado previamente, a su vez que éste objeto es asignado al atributo "mascota" de la instancia de Empleado.

```
public void asignarVeterinario(Mascota mascota, Empleado veterinario) {
    mascota.setVeterinario(veterinario);
    veterinario.setMascota(mascota);
}
```

Una vez asignado el veterinario que atenderá a la mascota, el sistema registra al animal en la lista de animales hospitalizados de la sede.

Cuando la atención médica o la hospitalización se han confirmado, el sistema gestiona el pago utilizando el método procesarPago() de CentroAdopcion. Las opciones disponibles incluyen: Tarjeta de crédito/débito, efectivo y/o puntos acumulados. Si el cliente elige pagar con puntos acumulados, el sistema valida el saldo de puntos disponibles mediante el método disminuirPuntos() de Cliente. Si los puntos no son suficientes, se solicita otro método de pago.

```java
public boolean procesarPago(int metodo, Cliente cliente, int monto) {
    switch (metodo) {
        case 1:
            System.out.println("\nPago procesado con tarjeta por un monto de: $" + monto);
            return true;
        case 2:
            System.out.println("\nPago procesado en efectivo por un monto de: $" + monto);
            return true;
        case 3:
            if (cliente != null && monto <= cliente.getPuntos()) {
                cliente.disminuir_Puntos(monto);
                System.out.println("\nPago procesado con puntos acumulados.");
                return true;
            } else {
                System.out.println("\nNo tiene suficientes puntos.");
                return false;
            }
        default:
            System.out.println("\nMétodo de pago no válido.");
            return false;
    }
}
```

```java
public void disminuir_Puntos(int puntos) {
    this.puntos-=puntos;
}
```

Al completar el pago, el sistema genera una factura detallada mediante el método generarFactura() de la clase CentroAdopcion. Esta factura incluye: Datos del cliente, información del animal y el monto total. La factura se presenta al usuario como comprobante del proceso.

```java
public String generarFactura(Cliente cliente, Mascota mascota, int monto) {

    return "\n--------- Factura --------- " + "\n*|* Cliente     *|* " + (cliente != null ? cliente : "No registrado") +
           "\n*|* Animal      *|* " + mascota + "\n*|* Monto total *|* " + monto + "\n-----------------------------\n";

}
```

Finalmente, se le notificará al usuario que su mascota puede ser dada de alta, en caso de alta médica, el sistema permite registrar el alta a través del método registrarAlta(), el cual elimina la mascota de la lista de animales hospitalizados en el centro de adopción, cambia el atributo “estadoSalud” de la mascota a “sano”, y desvincula al veterinario de la mascota y viceversa. Por otro lado, si el animal no es hospitalizado, el usuario será redirigido a Planificación de Dieta.

```java
public void registrarAlta(Mascota mascota) {
    mascota.setEstadoSalud(EstadoSalud.SANO);
    animalesHospitalizados.remove(mascota);
    mascota.getVeterinario().setMascota(null);
    mascota.setVeterinario(null);
}
```

Diagrama de flujo

- **FUNCIONALIDAD 2: Agendar: UNServicio**

El Centro Veterinario Virtual UNamascota ofrece una funcionalidad para agendar UNServicio de forma semanal para diversos servicios, con el objetivo primordial de garantizar el bienestar y cuidado de las mascotas.

Esta herramienta permite a los usuarios solicitar y programar citas fácilmente para acceder a

los servicios especializados que se ofrecen en cada una de las sedes.

Servicios ofrecidos

- En la sede Medellín, el servicio de Entrenamiento y Veterinaria.
- En la sede Bogotá, el servicio de Peluquería.
- En la sede de Cali, el servicio de Entrenamiento y Veterinaria.
- En la sede de Cartagena, el de Entrenamiento.

Los usuarios pueden agendar citas semanales con estos pasos:

- Elegir la sede y el correspondiente servicio
- Seleccionar el día y horario disponible de la semana
- Proporcionar información básica del cliente y de la mascota.
- Confirmar la cita, que se agrega automáticamente al calendario

La agenda de citas se actualiza dinámicamente para evitar conflictos y permite un máximo de citas por día según la capacidad de cada sede.

La ejecución de la funcionalidad comienza con el cliente eligiendo la sede, dependiendo del servicio que requiera.

A continuación se le mostrará los tipos de mascotas para los cuales ese servicio está disponible; se le consultará si el tipo de su mascota, coincide con los ofertados; de no coincidir, se le informará que el proceso de agendar el servicio no podrá continuar.

De lo contrario; en primer lugar se verificará que la sede cuente con empleados con cupos disponibles en su agenda para atender a las mascotas, esto se ejecuta mediante el método tieneEmpleados() de la clase CentroAdopcion el cual retornará un ArrayList con los empleadoDisponibles; el método en su lógica recorrera el ArrayList de empleados de la sede selccionada y en cada Empleado ejecuta el método tieneCupos() de su misma clase.

```
def actualizar_datos(self):
    for cupos_dia in self.agenda_dias:
        Cupo.actualizar_cupo(cupos_dia)

def tiene_cupos(self) -> bool:
    self.actualizar_datos()
    return any(cupo.is_disponible() for cupos_dia in self.agenda_dias for cupo in cupos_dia)

def cupos_disponibles(self, dia: int) -> List[Cupo]:
    disponibles = []
    for cupos_dia in self.agenda_dias:
        if cupos_dia[0].get_dia().weekday() == dia:
            disponibles.extend(cupo for cupo in cupos_dia if cupo.is_disponible())
            break
    return disponibles
```

El método tieneCupos(), en su lógica, lo primero que hará es actualizar los cupos del empleado desde el día actual hasta los próximos cinco días, mediante el método actualizarDatos() que habilita los cupos de los días posteriores, seguidamente lo que hará es recorrer el ArrayList de objetos de tipo cupo del empleado y verificar si alguno tiene disponibilidad con el metodo del la clase Cupo actualizarCupo().

El valor resultante, que es un boolean, lo tomará el método tieneEmpleado() para saber si

agregar o no al empleado en el ArrayList de empleadosDisponibles que será retornado al final del método.

```
def tieneEmpleados(self):
    return [empleado for empleado in self._veterinarios if empleado.tiene_cupos()]
```

Si el ArrayList que se retorna a la clase main, está vacío, es porque en la sede no hay empleados con disponibilidad para atender citas, por lo cual se le informará que el proceso no podrá continuar por falta de disponibilidad. De no ser así, es porque existen empleados con disponibilidad para atender a las mascotas.

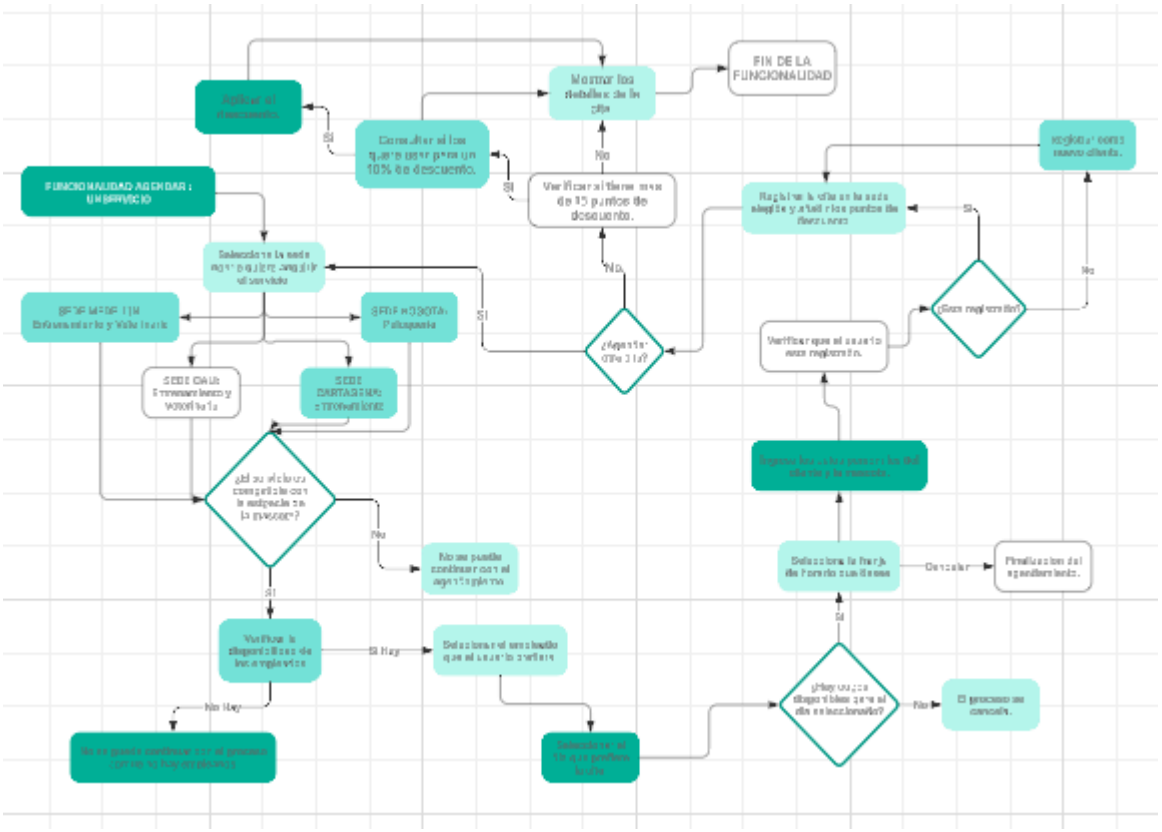
Hecho esto, el usuario tendrá la posibilidad de escoger el empleado de su preferencia entre los disponibles; posterior a esto podrá seleccionar el día en el que desea agendar la cita, cabe mencionar que las citas se agendan semanalmente, es decir desde el día actual, hasta los próximos cinco días, sin incluir el domingo. Después de seleccionar el día, al empleado seleccionado se le aplicará el método cupos_disponibles() el cual recibirá un entero correspondiente al día de la semana, siguiendo el orden, uno para lunes, dos para martes, tres para miércoles y consecutivamente.

```
    def tiene_cupos(self) -> bool:
        self.actualizar_datos()
        return any(cupo.is_disponible() for cupos_dia in self.agenda_dias for cupo in cupos_dia)
```

El método retornará un ArrayList con los cupos que tengan disponibilidad en el día seleccionado por el usuario, si el ArrayList que retorna, está vacío, se le informará al cliente que no se puede continuar con el proceso agendar servicio a falta de disponibilidad de cupos por parte del empleado. De otro modo, el usuario podrá seleccionar el cupo que sea de su grado en el horario ofertado.

Si todo sale bien al momento de generar una cita, se le informará un mensaje de que su cita fue agendada.

DIAGRAMA DE INTERACCIÓN:



https://lucid.app/lucidchart/6a9620a0-baf6-4287-a115-479b6a095fe8/edit?viewport_loc=-684%2C39%2C3570%2C1407%2C0_0&invitationId=inv_50023d5e-46b7-499d-b750-55da22dc0b9d

- **FUNCIONALIDAD 3:** Tienda UNamascota

En el main, podemos llamar al método iniciar_tienda(), cuyo funcionamiento consiste en crear un objeto de tipo Tienda y anexar a una sede. Una vez creado, el objeto de tipo Tienda comienza con una lista de objetos de tipo Producto (que tienen los atributos de nombre, tipo de animal, tipo de uso, precio y unidades). Los productos de esta lista son los que se ofertarán en la tienda.

Se llama al método comprar que despliega interacciones con el usuario según cómo desea visualizar sus productos

```
def inicio_tienda():
    limpiar_frame(content_frame)

    # Cargar productos desde el archivo serializado
    productos = cargar_datos_productos()

    if not productos:
        messagebox.showwarning("Error", "No se encontraron productos disponibles.")
        return
```

Después de este primer proceso, en el método tienda() de la clase main, se le pregunta al usuario: ¿Cómo desea que se le muestren los productos? El usuario tiene dos opciones: filtrar por el tipo de animal al que va dirigido el producto o mostrar todos los productos. Dependiendo de la respuesta, se evalúa en un condicional, mostrar_todo() o filtrar_por_tipo(), retorna la lista con los productos que el usuario desea ver.

```
def comprar():
    limpiar_frame(content_frame)
    ttk.Label(content_frame, text="¿Cómo le gustaría mostrar los productos?", font=("Arial", 14)).pack(pady=10)

    def mostrar_todo():
        limpiar_frame(content_frame)
        ttk.Label(content_frame, text="Seleccione un producto:", font=("Arial", 14)).pack(pady=10)
```

```
def filtrar_por_tipo():
    limpiar_frame(content_frame)
    ttk.Label(content_frame, text="Seleccione el tipo de animal:", font=("Arial", 14)).pack(pady=10)

    tipo_animal_var = tk.StringVar()
    tipos_animal = ["Perro", "Gato", "Conejo", "Ave"]
    tipo_combobox = ttk.Combobox(content_frame, textvariable=tipo_animal_var, values=tipos_animal, state="readonly")
    tipo_combobox.pack(pady=10)
```

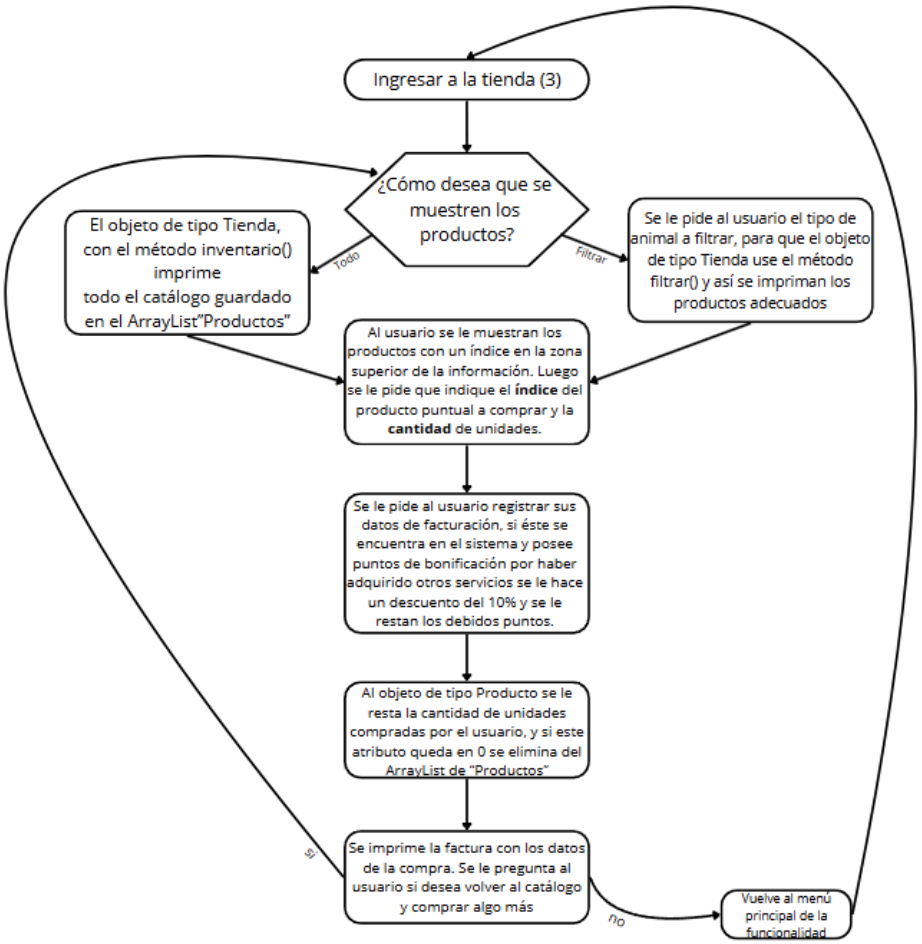
Una vez se muestran los productos al usuario, se le pide que seleccione el producto que desea comprar. Acto seguido, se le solicita la cantidad de unidades que quiere adquirir, y luego se vuelve a llamar a nuestro objeto Tienda. Este, con su método compra(), se encarga de restar la cantidad de unidades del producto solicitado y, posteriormente, retorna una pequeña factura con el monto a pagar.

```
def ingresar_datos_cliente(producto, cantidad):
    limpiar_frame(content_frame)

    ttk.Label(content_frame, text="Ingrese sus datos:", font=("Arial", 14)).pack(pady=10)

    ttk.Label(content_frame, text="Nombre:", font=("Arial", 12)).pack(pady=5)
    nombre_entry = ttk.Entry(content_frame)
    nombre_entry.pack(pady=5)

    ttk.Label(content_frame, text="Cédula:", font=("Arial", 12)).pack(pady=5)
    cedula_entry = ttk.Entry(content_frame)
    cedula_entry.pack(pady=5)
```

```
        total = producto.precio * cantidad
        recibo = f"""
        Recibo de Compra:
        Producto: {producto.nombre}
        Cantidad: {cantidad}
        Precio Unitario: ${producto.precio}
        Total: ${total}

        Datos del Cliente:
        Nombre: {nombre}
        Cédula: {cedula}
        """
```

Si la tienda detecta que, al momento de hacer la compra, el usuario está registrado (con el método compra() que recibe también un objeto de tipo Cliente) o ha sido comprador de alguno de los servicios, y además cuenta con más de 15 puntos de bonificación, recibirá un descuento del 10% en su compra.

Finalmente, es importante tener en cuenta que la tienda tiene su propia lista de empleados, y necesita que al menos un objeto de tipo Empleado esté en esta lista para poder realizar las acciones de mostrar productos (método mostrar_todo() o filtrar_por_tipo()).



- FUNCIONALIDAD 4: Memorial para mascotas:

En el **main** podemos llamar al método **gestionarMemorial(),** el cual crea un objeto de tipo Memorial.

La funcionalidad primero pide los datos del usuario en caso de querer crear un memorial para su mascota difunta. Luego se despliega un menú con las opciones:

1. Añadir memorial: Se encarga de crear un objeto de tipo **Fallecido**, que lleva en los atributos los detalles que se mostrarán en el memorial creado, como: nombre de la mascota, fecha de fallecimiento y un mensaje dejado por su dueño.

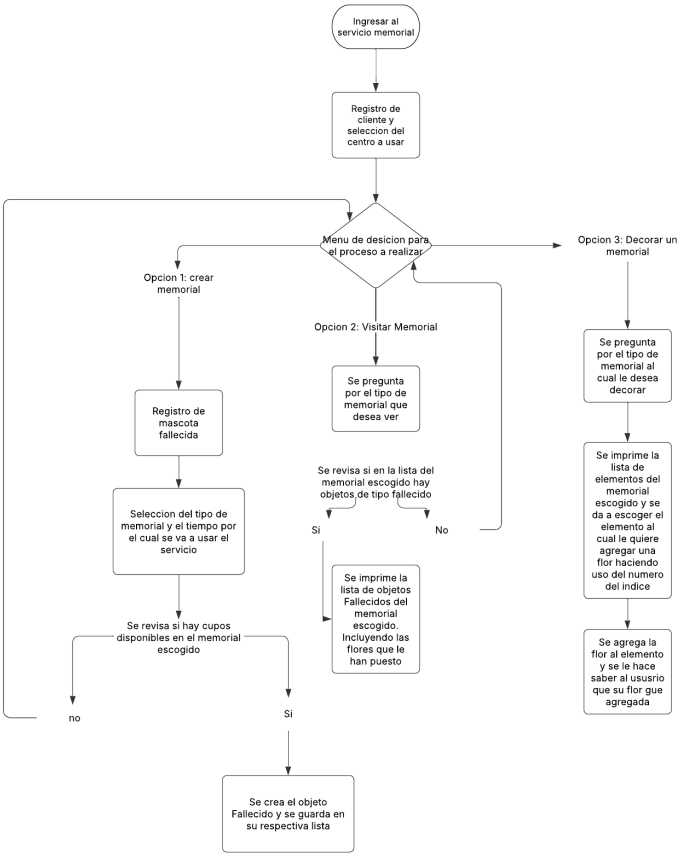
Luego de haber creado el objeto, se guarda en una lista dentro del objeto **Memorial**

con el respectivo tipo de memorial que se desee crear para la difunta mascota (Cenizas, Osario, Tumba o Plantar un árbol). El **main** luego preguntará por el tiempo durante el cual se desea emplear el servicio memorial, teniendo la opción de usar el memorial de por vida o por un tiempo limitado (se puede escoger años en múltiplos de 5). Independientemente de la acción, se imprimirá en pantalla el monto a pagar por el uso del memorial. En caso de que el proceso haya salido de manera correcta, el **main** imprimirá un mensaje informando al dueño que el proceso fue exitoso. De lo contrario (si el memorial escogido no tiene cupos), se le hará saber al dueño que no hay cupos suficientes y el proceso de creación del memorial terminará.

2. Ver memorial: El **main** se encargará de preguntar qué tipo de memorial desea visitar (Cenizas, Osario, Sepulcro o los árboles plantados) perteneciente al objeto Memorial. Dependiendo del tipo de memorial que se desee visitar, se hace uso del método **obtenerFallecidosPorTipo()**, que recibe un parámetro tipo y selecciona la lista respectiva al tipo de memorial que se seleccionó (las listas hacen parte del objeto **Memorial: ArrayList<Fallecido> tipo**). Luego, se hace uso del método **visitaMemorial()**, el cual imprime el índice (+1) seguido de los objetos Fallecido que forman parte de la lista seleccionada, incluyendo las decoraciones que se le han dejado.

3. Decorar memorial: El **main** preguntará el tipo de memorial que se quiere decorar (Cenizas, Osario, Sepulcro o los árboles plantados). Luego, se hace uso del método **obtenerFallecidosPorTipo()**, que imprimirá los elementos de la lista del tipo seleccionado indicados por un índice. Dicho índice puede ser usado por el usuario para escoger el objeto de tipo Fallecido al cual quiere dejar flores. El método **ponerFlor()** se encarga de guardar en un **ArrayList<>** todas las flores que se le han dejado al objeto de tipo **Fallecido**.

Diagrama de flujo funcionalidad 4:



https://lucid.app/lucidchart/8d17bf81-0bb4-4795-9c4b-e65cfbdce62c/edit?viewport_loc=109%2C939%2C2244%2C1053%2C0_0&invitationId=inv_89fecb3b-8155-4d2b-8cf9-a4a5d2c9b66b

- FUNCIONALIDAD 5: Planificación de Dietas para Mascotas:

En el MainWin, se llama al comando mostrar_formulario_dietas() a traves de un boton de tk, que tiene como objetivo capturar los datos de la mascota para generar un plan alimenticio personalizado utilizando calcular_dieta()

Fases de la funcionalidad:

1. Registro de la Mascota

Una vez iniciado el método, el sistema solicita los datos la mascota, su nombre, especie, edad, sexo, tamaño y peso (kilos). Solo se aceptaran perros y gatos, los tamaños consisten en Miniatura, Pequeño, Mediano, Grande El sistema valida que la especie ingresada sea válida y que los datos sean coherentes antes de continuar.

```
#Formulario para crear el objeto Mascota

tk.Label(content_frame, text="Planificación de Dieta", font=("Arial", 18, "bold")).pack(pady=10)

tk.Label(content_frame, text="Nombre de la mascota:").pack()
entry_nombre = tk.Entry(content_frame)
entry_nombre.pack()

tk.Label(content_frame, text="Especie:").pack()
especie_var = tk.StringVar()
opciones_especie = ["Perro", "Gato"]
tk.OptionMenu(content_frame, especie_var, *opciones_especie).pack()

tk.Label(content_frame, text="Edad (años):").pack()
entry_edad = tk.Entry(content_frame)
entry_edad.pack()

tk.Label(content_frame, text="Sexo:").pack()
sexo_var = tk.StringVar()
opciones_sexo = ["Macho", "Hembra"]
tk.OptionMenu(content_frame, sexo_var, *opciones_sexo).pack()

tk.Label(content_frame, text="Tamaño:").pack()
tamano_var = tk.StringVar()
opciones_tamano = ["Miniatura", "Pequeño", "Mediano", "Grande"]
tk.OptionMenu(content_frame, tamano_var, *opciones_tamano).pack()

tk.Label(content_frame, text="Peso en kg:").pack()
entry_peso = tk.Entry(content_frame)
entry_peso.pack()
especie = especie_var.get()
```

2. Cálculo del Peso Ideal

Después del registro, se crea un objeto Mascota con los datos introducidos, se invoca el método calcularPesoIdeal(), el cual evalúa el peso ideal de la mascota utilizando la fórmula:

```
# Crear objeto Mascota
mascota = Mascota(nombre, especie, edad, sexo, EstadoSalud.SANO , tamano, peso)

# Crear y calcular dieta
dieta = Dieta(mascota)
dieta.calcularPesoIdeal()
dieta.planDieta()
dieta.menu()
messagebox.showinfo(f"Plan de dieta", f"{dieta}")
```

$$Peso\ ideal\ = \frac{tamaño \times 10}{edad + 1} + \frac{peso\ actual}{2}$$

El método comparacionPeso() compara su peso ideal con el peso actual y determinará si la mascota debe subir o bajar de peso.

```
Métodos de cálculo
ef calcularPesoIdeal(self):
   tamano = self.mascota.tamano
   edad = self.mascota.edad
   pesoActual = self.mascota.peso
   self.pesoIdeal = ((tamano * 10.0) / (edad + 1)) + (pesoActual /

ef comparacionPeso(self):
   if self.mascota.peso == self.pesoIdeal:
       return 1  # Peso ideal
   elif self.mascota.peso < self.pesoIdeal:
       return 2  # Necesita subir de peso
   else:
       return 3  # Necesita bajar de peso
```

3. Planificación de la Dieta

El método planDieta() determinará la cantidad diaria de gramos de alimento que necesita la mascota utilizando la fórmula:

$$Gramos\ diarios\ =\ Peso\ Ideal\ \times\ Tamaño\ \times\ 10$$

y después utilizando el método comparacionPeso() determinará la distribución de proporciones de nutrientes:
Si necesita subir de peso: 30% grasas, 40% proteínas, 30% carbohidratos.

Si necesita bajar de peso: 15% grasas, 50% proteínas, 35% carbohidratos.

Si está en su peso ideal: 20% grasas, 30% proteínas, 50% carbohidratos.

El sistema calcula la cantidad diaria de alimento necesaria y la distribuye en gramos de proteínas, carbohidratos y grasas.

```
def planDieta(self):
    self.gramosDiarios = self.pesoIdeal * self.mascota.tamano * 10
    comparacion = self.comparacionPeso()

    if comparacion == 1:  # Peso ideal
        self.grasas = self.gramosDiarios * 0.20
        self.proteinas = self.gramosDiarios * 0.30
        self.carbohidratos = self.gramosDiarios * 0.50
    elif comparacion == 2:  # Subir de peso
        self.grasas = self.gramosDiarios * 0.30
        self.proteinas = self.gramosDiarios * 0.40
        self.carbohidratos = self.gramosDiarios * 0.30
    elif comparacion == 3:  # Bajar de peso
        self.grasas = self.gramosDiarios * 0.15
        self.proteinas = self.gramosDiarios * 0.50
        self.carbohidratos = self.gramosDiarios * 0.35
```

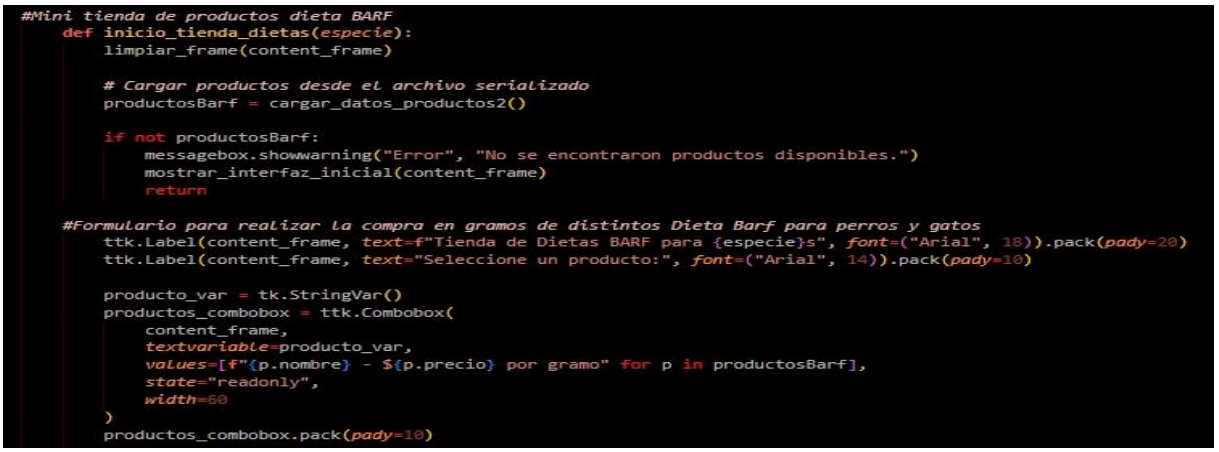
4. Generación del Menú Nutricional

El método toString (o **__str__**) crea un menú sugerido basado en los productos disponibles en la tienda, y un pequeño resumen del estado físico de la mascota. Además, el método menu() creará un archivo .txt que guardará lo que se imprima con toString().

```
def __str__(self):
    estadoPeso = {
        1: "Está en su peso ideal.",
        2: "Debe subir de peso.",
        3: "Debe bajar de peso."
    }.get(self.comparacionPeso(), "Estado no definido.")

    tamanoStr = ""
    if self.mascota.tamano == 1:
        tamanoStr = "Miniatura"
    elif self.mascota.tamano == 2:
        tamanoStr = "Pequeño"
    elif self.mascota.tamano == 4:
        tamanoStr = "Grande"
    else:
        tamanoStr = "Mediano"

    return (f"Nombre de la mascota: {self.mascota.nombre}\n"
            f"Peso Actual: {self.mascota.peso} kg\n"
            f"Edad: {self.mascota.edad} años\n"
            f"Tamaño: {tamanoStr}\n"
            f"Peso ideal: {round(self.pesoIdeal, 2)} kg\n"
            f"Cantidad de Gramos de alimento diarios: {round(self.gramosDiarios, 2)} g\n"
            f"{estadoPeso}\n\n"
            f"Distribución en porcentajes de nutrientes:\n"
            f" Proteínas: {round((self.proteinas / self.gramosDiarios) * 100, 2)}%\n"
            f" Grasas: {round((self.grasas / self.gramosDiarios) * 100, 2)}%\n"
            f" Carbohidratos: {round((self.carbohidratos / self.gramosDiarios) * 100, 2)}%\n\n"
            f"Distribución en gramos de nutrientes:\n"
            f" Proteínas: {round(self.proteinas, 2)} g\n"
            f" Grasas: {round(self.grasas, 2)} g\n"
            f" Carbohidratos: {round(self.carbohidratos, 2)} g")
```

5. Integración con la Tienda Virtual

Después de la generación de la dieta, el usuario tiene la opción de adquirir Dieta Barf para su mascota en la tienda virtual. El sistema determinará si se necesita Dieta Barf para gato o perro, y presentará el catálogo de alimentos disponibles a granel.



```
#Mini tienda de productos dieta BARF
    def inicio_tienda_dietas(especie):
        limpiar_frame(content_frame)

        # Cargar productos desde el archivo serializado
        productosBarf = cargar_datos_productos2()

        if not productosBarf:
            messagebox.showwarning("Error", "No se encontraron productos disponibles.")
            mostrar_interfaz_inicial(content_frame)
            return

    #Formulario para realizar la compra en gramos de distintos Dieta Barf para perros y gatos
        ttk.Label(content_frame, text=f"Tienda de Dietas BARF para {especie}s", font=("Arial", 18)).pack(pady=20)
        ttk.Label(content_frame, text="Seleccione un producto:", font=("Arial", 14)).pack(pady=10)

        producto_var = tk.StringVar()
        productos_combobox = ttk.Combobox(
            content_frame,
            textvariable=producto_var,
            values=[f"{p.nombre} - ${p.precio} por gramo" for p in productosBarf],
            state="readonly",
            width=60
        )
        productos_combobox.pack(pady=10)
```

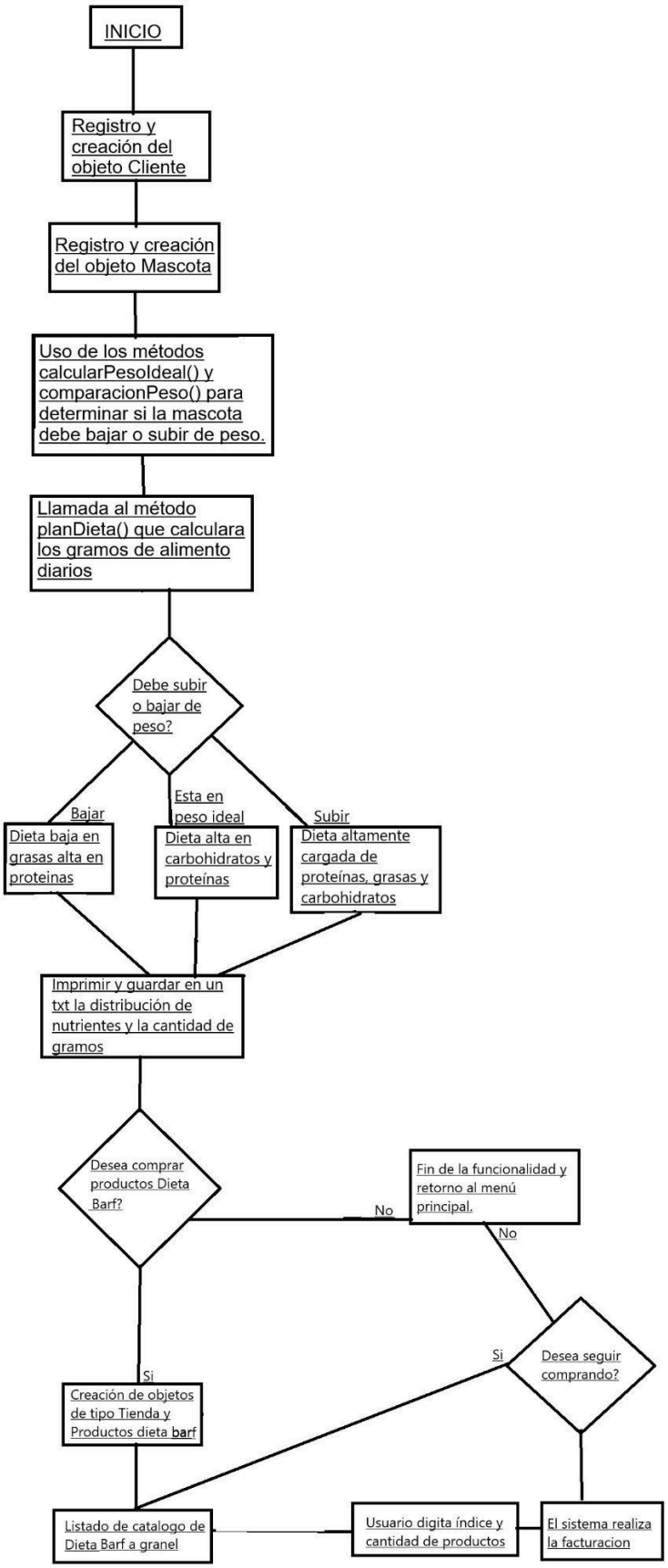
El usuario elige un producto y la cantidad en gramos que desea comprar y se realiza la factura electrónica.

```
#Consumacion de la compra y factura electronica.
def confirmar_compra():
    producto_seleccionado = producto_var.get()
    cantidad = int(cantidad_var.get())
    nombre = nombre_entry.get()
    cedula = cedula_entry.get()

    if not producto_seleccionado or cantidad <= 0 or not nombre or not cedula:
        messagebox.showwarning("Error", "Por favor complete todos los campos correctamente.")
        return

    nombre_producto = producto_seleccionado.split(" - ")[0]
    producto = next((p for p in productosBarf if p.nombre == nombre_producto), None)

    if producto:
        cliente = Cliente(nombre,18, cedula)
        messagebox.showinfo("Factura Electrónica",
                            f"FACTURA ELECTRONICA\n{producto.nombre} x {cantidad}g.\nCliente: {cliente.nombre}\nDocumento: {cliente.cedula}\nMuchas Gracias por comprar en UNmascota! Vuelva pronto")
        mostrar_interfaz_inicial(content_frame)
    else:
        messagebox.showwarning("Error", "Producto no encontrado.")
```

6. Finalización y Redirección

Al finalizar, el sistema muestra un mensaje de confirmación con la factura digital y redirige al usuario al menú principal.

Diagrama de Interacción:



7. **FieldFrame.**

La clase FieldFrame es un componente visual basado en Frame de la librería tkinter, diseñado para estructurar y administrar etiquetas junto con campos de entrada en una interfaz gráfica. Su objetivo principal es proporcionar una manera eficiente de visualizar y modificar datos organizados en un formato de filas y columnas, facilitando la interacción del usuario con formularios dinámicos.

Al crear una instancia de FieldFrame, es posible establecer títulos para las columnas de criterios y valores, así como definir una lista de criterios que representen los nombres de los campos. De manera opcional, se pueden asignar valores predeterminados y especificar qué campos deben estar deshabilitados mediante una lista de valores booleanos. Estos parámetros se almacenan como atributos de la instancia y se utilizan para generar la interfaz de manera dinámica.

El diseño de FieldFrame sigue un esquema bien definido. Primero, se ubican los títulos de las columnas en la parte superior. Luego, se recorren los criterios para generar una etiqueta y un campo de entrada por cada uno. Si hay valores predefinidos, estos se insertan automáticamente en los campos, y si se ha indicado que algún campo debe estar deshabilitado, se configura en modo de solo lectura.

Además de encargarse de la creación de estos elementos, la clase incorpora métodos para obtener (getValue) y actualizar (setValue) el contenido de los campos, así como para deshabilitar entradas específicas (disableEntry). También permite añadir botones con funciones personalizadas mediante crearBotones y sustituir un campo de entrada por un Combobox con el método agregar_combobox, aumentando así la versatilidad del formulario.

En cuanto a su utilidad dentro de una aplicación con interfaz gráfica, FieldFrame se convierte en un componente reutilizable que simplifica la construcción y gestión de formularios estructurados. Su implementación optimiza la administración de datos introducidos por el usuario, garantizando una organización clara y funcional de la información.

```python
from tkinter import *
from tkinter import ttk
class FieldFrame(Frame):
    def __init__(self, ventana, tituloCriterios="", criterios=[], tituloValores="", valores=None, habilitado=None):
        super().__init__(ventana)
        self._tituloCriterios = tituloCriterios
        self._criterios = criterios
        self._tituloValores = tituloValores
        self._valores = valores
        self._habilitado = habilitado
        self._elementos = []

        # Configuración de estilo
        self.configure(bg='#f7f7f7')  # Fondo más claro
        estilo_etiqueta = {'font': ("Poppins", 12, 'bold'), 'bg': '#f7f7f7', 'fg': '#333'}
        estilo_campo = {'font': ("Poppins", 11), 'bg': '#ffffff', 'fg': '#333', 'bd': 2, 'relief': GROOVE}

        # Crear y colocar titulo de los criterios
        etiquetaTituloCriterios = Label(self, text=tituloCriterios, font=("Poppins", 14, 'bold'), bg='#f7f7f7', fg='#000'
        etiquetaTituloCriterios.grid(column=1, row=0, padx=(10, 10), pady=(10, 10), sticky='ew')

        # Crear y colocar titulo de los valores
        etiquetaTituloValores = Label(self, text=tituloValores, font=("Poppins", 14, 'bold'), bg='#f7f7f7', fg='#000')
        etiquetaTituloValores.grid(column=2, row=0, padx=(10, 10), pady=(10, 10), sticky='ew')

        # Crear y colocar las etiquetas y campos de entrada para cada criterio
        for i in range(len(criterios)):
            etiquetaCriterio = Label(self, text=criterios[i], **estilo_etiqueta)
            etiquetaCriterio.grid(column=1, row=i + 1, padx=(10, 10), pady=(5, 5), sticky='e')

            campoValor = Entry(self, **estilo_campo)
            campoValor.grid(column=2, row=i + 1, padx=(10, 10), pady=(5, 5), sticky='ew')
```

```python
        return self._elementos[indice].get()

    def setValue(self, criterio, valor):
        indice = self._criterios.index(criterio)
        self._elementos[indice].delete(0, END)
        self._elementos[indice].insert(0, valor)

    def disableEntry(self, criterio):
        indice = self._criterios.index(criterio)
        self._elementos[indice].configure(state=DISABLED)

    def crearBotones(self, comando1, texto="Aceptar", pady=8, column=1, padx=5):
        estilo_boton = {
            'font': ("Poppins", 10, 'bold'),
            'fg': 'black',
            'bg': '#d3d3d3',  # Gris claro
            'activebackground': '#bfbfbf',
            'bd': 2,
            'relief': RAISED,
            'cursor': "hand2",
            'width': 12,
            'height': 1
        }
        Button(self, text=texto, command=comando1, **estilo_boton).grid(padx=padx, pady=pady, column=column, row=len(sel

    def agregar_combobox(self, criterio, valores):
        indice = self._criterios.index(criterio)
        cuadroOpciones = ttk.Combobox(self, values=valores, state="readonly", font=("Poppins", 10))
        cuadroOpciones.grid(column=2, row=indice + 1, padx=(10, 10), pady=(5, 5), sticky='ew')
        self._elementos[indice] = cuadroOpciones
```

Aplicaciones:

```python
def manejar_seleccion_proceso(proceso):
    limpiar_frame(content_frame)

    field_frame_proceso = FieldFrame(content_frame, "Proceso Seleccionado", [proceso], "")
    field_frame_proceso.pack(pady=10)
```

```
field_frame_tienda = FieldFrame(content_frame, "Bienvenido a la Tienda", [], "")
field_frame_tienda.pack(pady=20)

def comprar():
    limpiar_frame(content_frame)
    field_frame_comprar = FieldFrame(content_frame, "¿Cómo le gustaría mostrar los productos?", [], "")
    field_frame_comprar.pack(pady=10)

    def mostrar_todo():
        limpiar_frame(content_frame)
        field_frame_todo = FieldFrame(content_frame, "Seleccione un producto", ["Producto", "Cantidad"], "Valor")
        field_frame_todo.pack(pady=10)

        productos_valores = [f"{p.nombre} - ${p.precio} ({p.tipo_animal})" for p in productos]
        field_frame_todo.agregar_combobox("Producto", productos_valores)
        field_frame_todo.setValue("Cantidad", 1)
```

```
    def inicio_tienda_dietas(especie):
        limpiar_frame(content_frame)

        # Cargar productos desde el archivo serializado
        productosBarf = cargar_datos_productos2()

        if not productosBarf:
            messagebox.showwarning("Error", "No se encontraron productos disponibles.")
            mostrar_interfaz_inicial(content_frame)
            return

        field_frame_tienda = FieldFrame(content_frame, f"Tienda de Dietas BARF para {especie}s", ["Producto", "Cantidad
        field_frame_tienda.pack(pady=10)
        field_frame_tienda.agregar_combobox("Producto", [f"{p.nombre} - ${p.precio} por gramo" for p in productosBarf])
```

Manual de Usuario - Aplicación UNaMascota

Introducción

Bienvenido a **UNaMascota**, una aplicación diseñada para gestionar diferentes aspectos del cuidado de tu mascota. A través de esta plataforma, podrás acceder a una tienda de productos dietéticos, agendar servicios, adquirir memoriales para mascotas fallecidas, generar planes de dieta personalizados y acceder a servicios de emergencia veterinaria.

Instrucciones Generales

Ventana de Inicio

Al abrir la aplicación, se mostrará una pantalla de inicio con imágenes y los nombres de los desarrolladores. Para acceder al programa principal, presiona el botón **"Ingresar"**.

- En la esquina superior izquierda, hay un botón **"Inicio"** con las opciones **"Acerca de"** y **"Salir"**.
- Al pasar el cursor sobre las imágenes de la izquierda, cambiarán a imágenes de los servicios disponibles.

Menú Principal

Desde el menú principal podrás acceder a las siguientes funcionalidades:

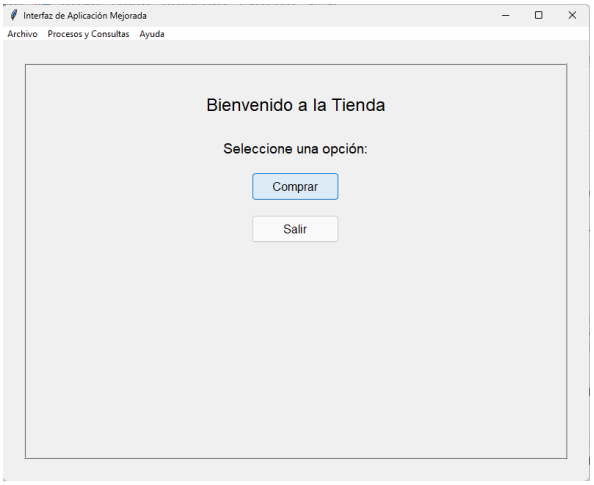
1. **Tienda** - Compra productos para tu mascota.
2. **Servicios** - Agendar citas en diferentes sedes.
3. **Emergencias** - Atención veterinaria inmediata.
4. **Memorial** - Crear y ver memoriales para mascotas fallecidas.
5. **Dieta** - Generar planes alimenticios y comprar dietas BARF.

Cada opción te llevará a una interfaz específica con sus respectivos formularios y opciones.

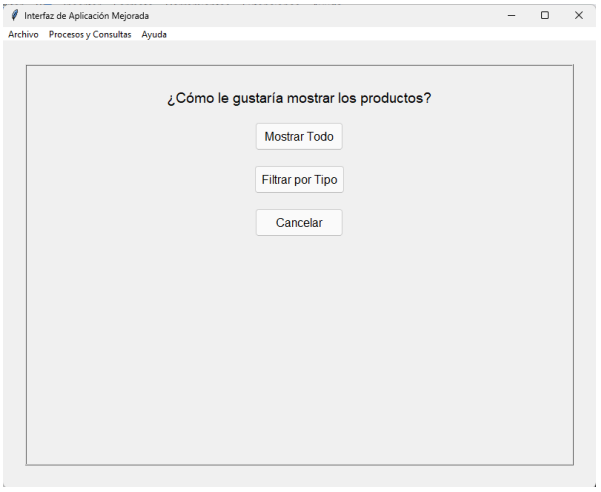
Tienda

En esta sección podrás comprar productos para tu mascota.

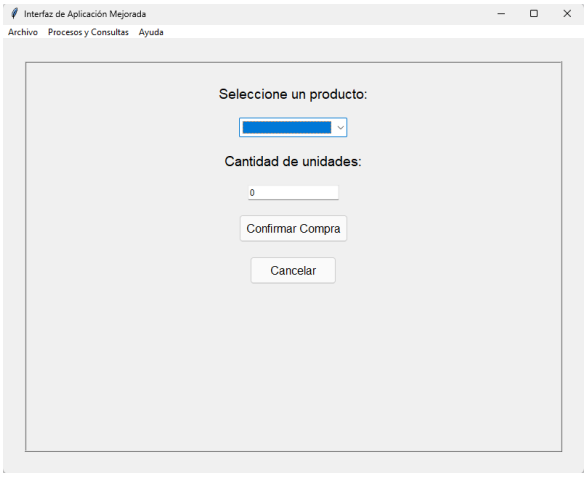
1. **Se le pide al usuario que escoja que desea hacer**
    - El usuario al entrar al Proceso 1: Tienda UNamascota puede seleccionar entre dos opciones: **“Comprar”** (El cual lo enviará a continuar con la funcionalidad) o **“Salir”** (El cual lo sacará del programa si no desea continuar el proceso)



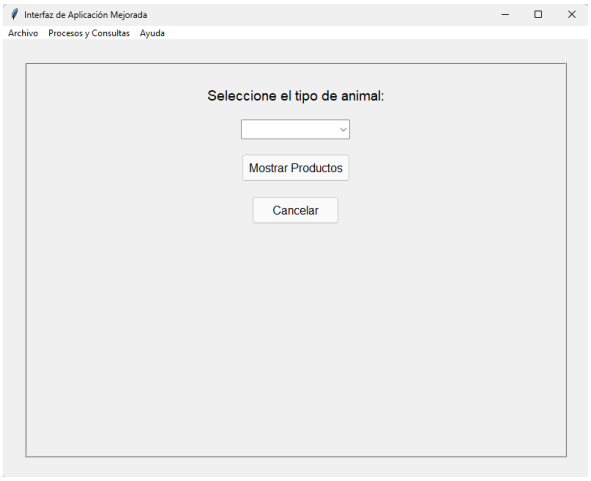
2. **Filtrar los tipos de productos**
    - Al continuar con el programa el usuario podrá escoger de qué modo se mostrarán los productos disponibles en la tienda



    - Si presiona el botón “Mostrar todo” se actualizará la ventana permitiéndote ver una lista que incluye todos los productos de la tienda.

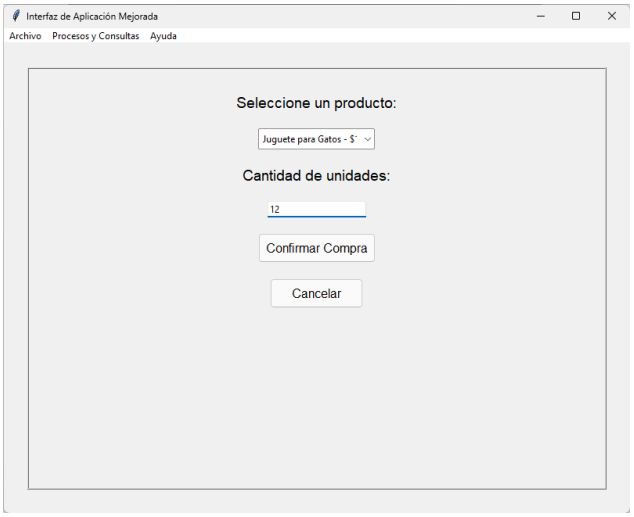


- Si presiona el botón “Filtrar por tipo” se le mostrará los tipos de productos que se tienen disponibles y a continuación se mostrarán los productos que cumplen con ese requisito de tipo. (Puede ser productos de: Perro, gato, conejo o ave)

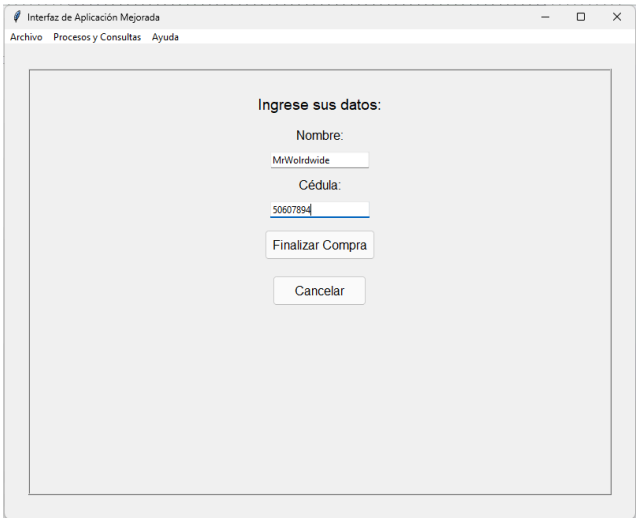


- Si presiona el botón “Cancelar” se terminará la funcionalidad de compra y saldrá de esta última.

3. **Realizar Compra**
   - Selecciona un producto luego de haber filtrado o haber visto todos los productos.
   - Indica la cantidad de unidades que desea del producto, si la cantidad del producto está disponible pasará al sistema de facturación.

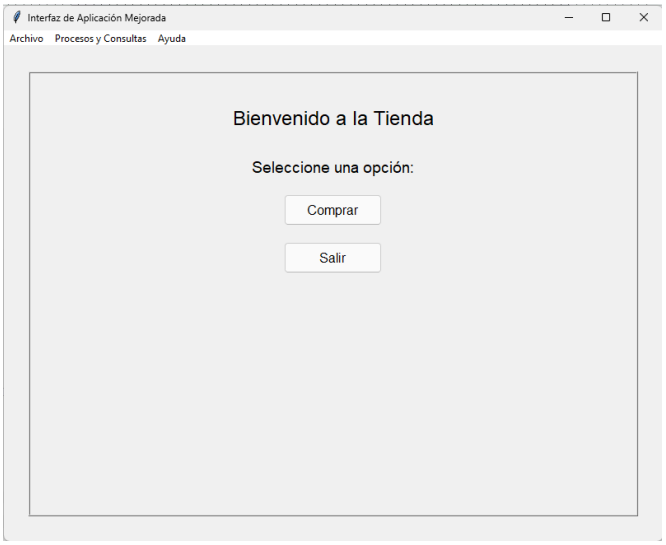


- En facturación le solicitaremos ingresar su nombre y su cédula.



- Al ingresar tus datos y seleccionar “Finalizar Compra” se mostrará en la ventana la información relacionada a su compra (Factura Digital), si seleccionas “Cancelar” volverás al inicio de la Tienda.

- Se le preguntará si desea volver al inicio, iniciando nuevamente el ciclo de la Tienda



Servicios

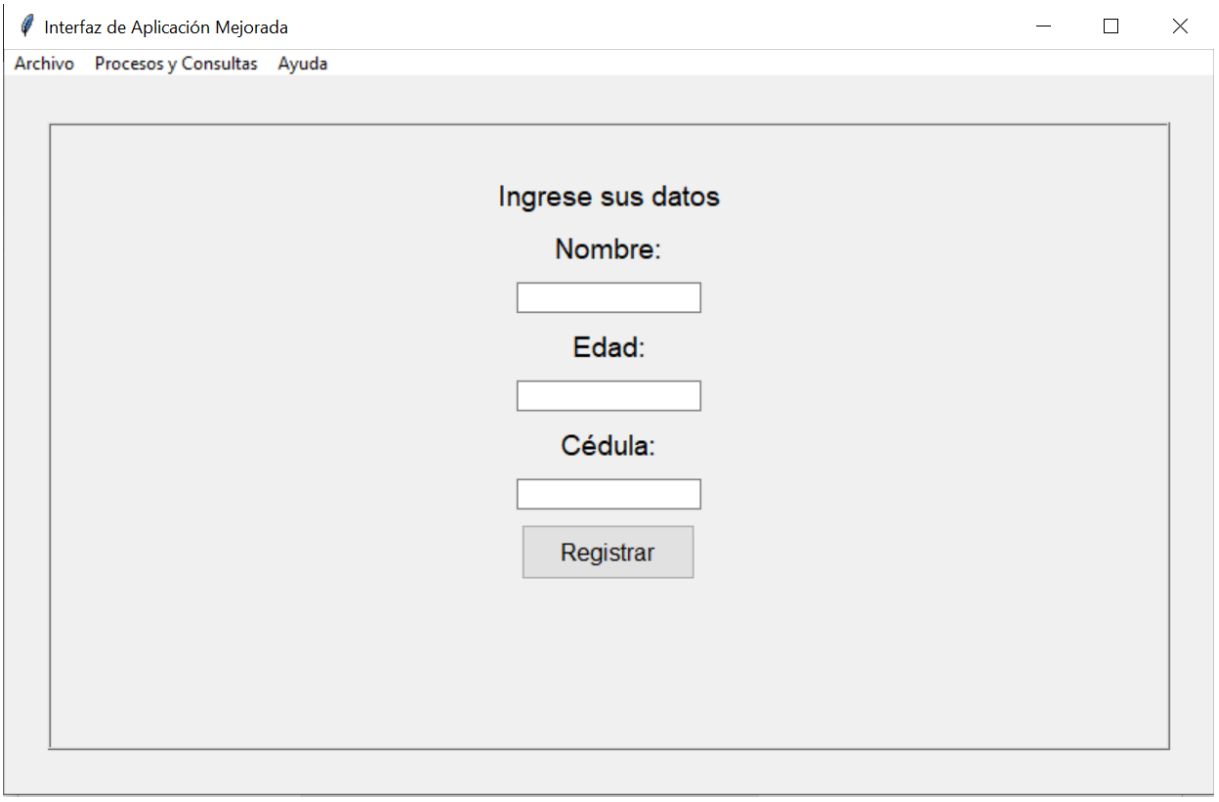
En esta sección podrás agendar citas para tu mascota en distintas sedes.

1. **Selecciona una sede:**
    - **Medellín:** Entrenamiento, Veterinaria.
    - **Bogotá:** Peluquería.
    - **Cali:** Entrenamiento, Veterinaria.
    - **Cartagena:** Entrenamiento.
2. **Selecciona el servicio y la raza de tu mascota** (Perro o Gato).
3. **Selecciona el empleado que deseas que te atienda**.
4. **Elige un día y una franja horaria**.
5. **Ingresa los datos tuyos y de tu mascota**.
6. **Confirma la cita** - Se mostrará un mensaje de confirmación en una ventana emergente.

Emergencias

En esta sección podrás acceder a atención veterinaria inmediata para tu mascota.

1. **Registro de usuario**
    - Ingresa:
        - Tu nombre.
        - Cédula.
        - Edad.



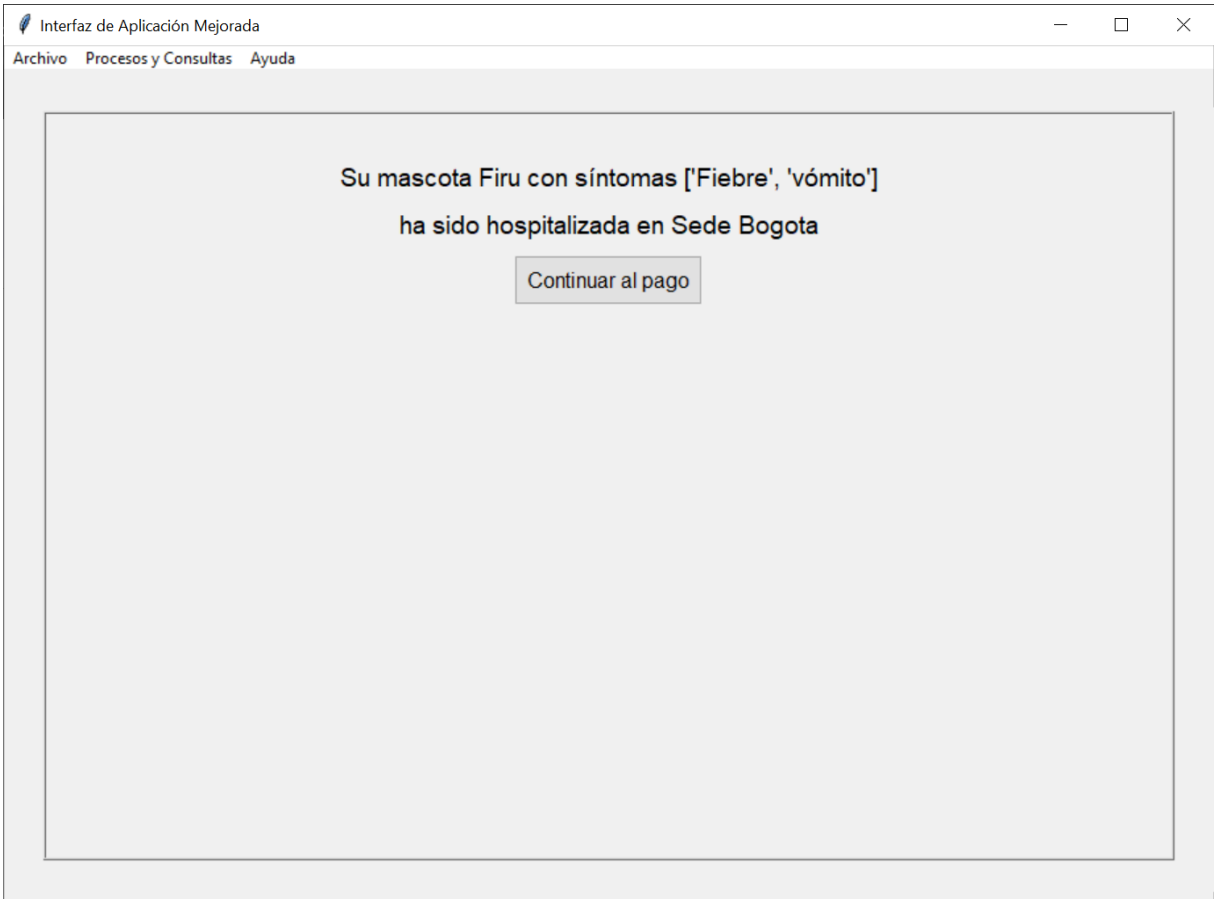
2. **Selecciona una sede de atención**
    - Medellín, Bogotá, Cali o Cartagena.

3. **Datos de la mascota**

    - Ingresa:
        - Nombre.
        - Especie (Perro o Gato).
        - Sexo.
        - Tamaño.
        - Peso en kilos.
        - Síntomas.

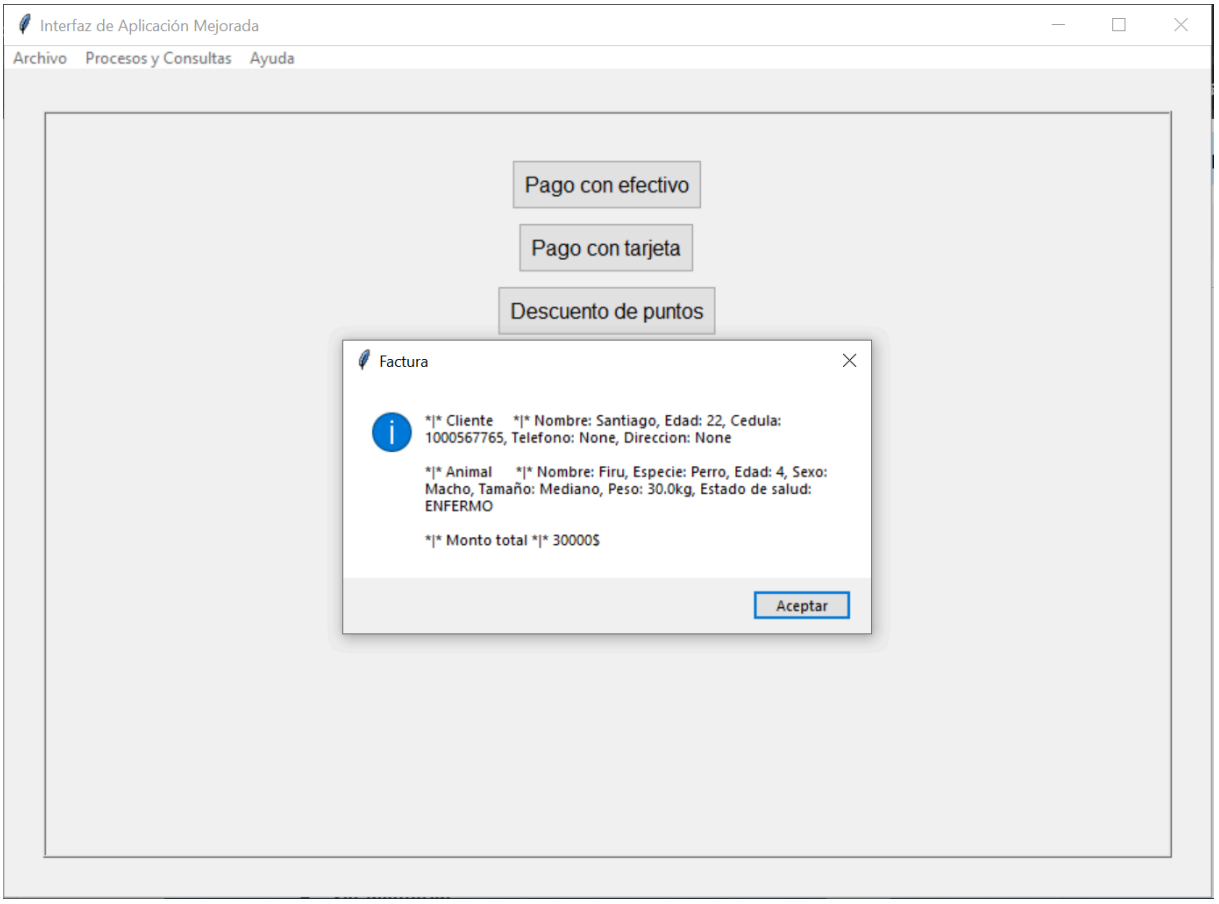
4. **Confirmación de hospitalización**

    - Se mostrará un mensaje indicando que tu mascota con los síntomas ingresados ha sido hospitalizada en la sede seleccionada.



5. **Pago del servicio**

    - Selecciona una de las opciones de pago:
        - **Efectivo:** Se generará una factura por **32000**.
        - **Tarjeta:** Se generará una factura por **30000**.
        - **Descuento de puntos:** Se generará una factura por **28000**.
    - Se mostrará la factura electrónica en una ventana emergente.
    - Al aceptar, regresarás al inicio de la funcionalidad de emergencias veterinarias.



Memorial

Esta sección permite crear y visualizar memoriales para mascotas fallecidas.

1. **Registro de usuario**
    - Ingresa tu nombre, edad y cédula.
    - Selecciona una de las siguientes opciones:
        - **Añadir Memorial**
        - **Ver Memorial**
        - **Decorar Memorial**
        - **Volver al Menú Principal**
2. **Añadir Memorial**

    - Ingresa:
        - Nombre de la mascota.
        - Especie.
        - Edad.
        - Fecha de fallecimiento.
        - Mensaje ("Descansa en paz", "Siempre en nuestros corazones", "Nunca te olvidaremos").
        - Tipo de memorial ("Tumba", "Osario", "Cenizas", "Árbol").

    - Guarda el memorial.
    - Un mensaje emergente confirmará que el memorial fue guardado correctamente.
3. **Ver Memorial**

    - Selecciona el tipo de memorial.
    - Se mostrará una lista con los memoriales guardados, incluyendo:
        - Nombre de la mascota.
        - Fecha de fallecimiento.
        - Frase seleccionada.
        - Flores decorativas añadidas.
4. **Decorar Memorial**

    - Selecciona el tipo de memorial y el número de registro.
    - Ingresa el nombre de la flor que deseas añadir.
    - Un mensaje emergente confirmará la decoración.

Dieta

Esta sección permite generar planes de alimentación y comprar dietas BARF.

1. **Generar Plan de Dieta**

    - Ingresa:
        - Nombre de la mascota.
        - Especie (Perro o Gato).
        - Edad.
        - Sexo.
        - Tamaño.
        - Peso en kilos.
    - Se generará un plan de dieta en una ventana emergente con:
        - Peso actual e ideal.
        - Cantidad de alimentos diaria.
        - Distribución de macronutrientes en porcentajes y gramos.
    - Se guardará un archivo .txt con la información en `/src/basedatos`.
2. **Comprar Dieta BARF**

    - Un mensaje emergente preguntará si deseas comprar una dieta BARF.
    - Si seleccionas **"No"**, regresarás al menú principal.
    - Si seleccionas **"Sí"**, podrás elegir entre:
        - **Alto en Carbohidratos**
        - **Alto en Proteínas**
        - **Alto en Grasas**
    - Ingresa:
        - Cantidad en gramos.
        - Tu nombre y tu cédula.
    - Confirma la compra.

- Se mostrará una factura electrónica en una ventana emergente.
- Al aceptar, regresarás al menú principal.

Conclusión

Este manual proporciona instrucciones detalladas sobre el uso de **UNaMascota**. Si tienes dudas o problemas, revisa las instrucciones de cada sección o consulta con soporte técnico. ¡Disfruta la aplicación!