Sistema de Gestión de Restaurante

Evaluación 2 — Programación II

Universidad Católica de Temuco
Facultad de Ingeniería
Ingeniería Civil Informática

Integrantes:
Joaquin Carrasco Duran

Benjamin Cabrera

Leonardo Chavez

Profesor: Guido Mellado

Asignatura: Programación II

Sección: 2

Octubre 2025

Índice

1. Introducción

Este informe presenta el desarrollo de un sistema de gestión para restaurantes implementado en Python. El sistema permite la administración de inventario, gestión de pedidos, generación de boletas y visualización de menús utilizando una interfaz gráfica moderna con customtkinter.

2. Objetivos

2.1. Objetivo General

Desarrollar un sistema de gestión integral para restaurantes que permita administrar inventario, pedidos y generación de documentos de manera eficiente.

2.2. Objetivos Específicos

- Implementar un sistema de gestión de inventario para ingredientes
- Crear un sistema de pedidos con interfaz gráfica
- Desarrollar un generador de boletas automatizado
- Implementar visualización de menús en formato PDF

3. Requerimientos Funcionales

3.1. Gestión de Inventario

- RF1: El sistema debe permitir agregar nuevos ingredientes al inventario
- RF2: El sistema debe permitir eliminar ingredientes existentes
- RF3: El sistema debe actualizar automáticamente las cantidades de ingredientes
- RF4: El sistema debe permitir cargar ingredientes desde archivos CSV
- RF5: El sistema debe validar la disponibilidad de ingredientes para menús

3.2. Gestión de Pedidos

- RF6: El sistema debe permitir agregar ítems del menú al pedido actual
- RF7: El sistema debe permitir eliminar ítems del pedido actual
- RF8: El sistema debe calcular automáticamente el total del pedido
- RF9: El sistema debe verificar la disponibilidad de ingredientes al agregar ítems
- RF10: El sistema debe actualizar el inventario al confirmar un pedido

3.3. Gestión de Menús

- RF11: El sistema debe mostrar el menú con imágenes representativas
- RF12: El sistema debe permitir generar una carta en formato PDF
- RF13: El sistema debe mostrar la disponibilidad de cada ítem del menú
- RF14: El sistema debe permitir visualizar el detalle de cada ítem

3.4. Generación de Documentos

- RF15: El sistema debe generar boletas con un identificador único
- RF16: El sistema debe calcular automáticamente el IVA (19 %)
- RF17: El sistema debe almacenar las boletas generadas en una carpeta específica
- RF18: El sistema debe permitir visualizar las boletas generadas en formato PDF

3.5. Interfaz de Usuario

- RF19: El sistema debe proporcionar una interfaz con pestañas para diferentes funcionalidades
- RF20: El sistema debe mostrar mensajes de confirmación para acciones importantes

- RF21: El sistema debe mostrar mensajes de error cuando ocurran problemas
- RF22: El sistema debe actualizar la interfaz en tiempo real al realizar cambios

4. Arquitectura del Sistema

El sistema está desarrollado siguiendo los principios de la programación orientada a objetos y utiliza varios patrones de diseño para mantener una estructura modular y mantenible.

4.1. Patrones de Diseño Utilizados

- **Patrón Facade**: Implementado en la clase BoletaFacade para simplificar la generación de boletas.
- **Protocol (Interfaz moderna)**: Utilizado en IMenu para definir el contrato de los elementos del menú. Se implementa usando el módulo typing.Protocol de Python, que proporciona una forma más flexible y moderna de definir interfaces.

  La implementación de IMenu utiliza Protocol en lugar de ABC (Abstract Base Class) para proporcionar un tipado estructural más flexible:

```
class IMenu(Protocol):
    """Interfaz para los elementos del menú utilizando tipado
    estructural."""
    nombre: str                         # Nombre del elemento del men
    ú
    ingredientes: List[Ingrediente]     # Lista de ingredientes
    requeridos
    precio: float                       # Precio del elemento
    cantidad: int                       # Cantidad en el pedido
    icono_path: Optional[str]           # Ruta al ícono (opcional)

    def esta_disponible(self, stock: Stock) -> bool:
        """Verifica disponibilidad en el stock dado."""
        ...
```

Listing 1: Interfaz IMenu usando Protocol

Esta implementación con Protocol permite:

- Tipado estructural: Las clases no necesitan declarar explícitamente que implementan IMenu
- Atributos tipados: Definición clara de tipos para cada atributo
- Compatibilidad implícita: Cualquier clase que tenga la estructura correcta es compatible
- Métodos abstractos: Define comportamiento requerido como `esta_disponible()`

- **Composición sobre Herencia**: El sistema favorece la composición. Por ejemplo, la clase `CrearMenu` no hereda de `Ingrediente`, sino que "se compone deüna lista de objetos `Ingrediente`, lo que resulta en un diseño más flexible.
- **Factory (Fábrica Simple)**: La función `get_default_menus()` actúa como una fábrica que centraliza la creación de los objetos de menú iniciales.
- **Immutable Object (Objeto Inmutable)**: La clase `CrearMenu` es inmutable (`frozen=True`), lo que previene modificaciones accidentales y promueve una gestión de estado más segura.
- **Observer (Observador Implícito)**: La GUI se actualiza llamando a métodos como `actualizar_treeview()` después de que los datos (el `Stock` o el `Pedido`) cambian, manteniendo la vista sincronizada con el modelo de datos.

5. Diagrama de Clases

5.1. Estructura del Sistema

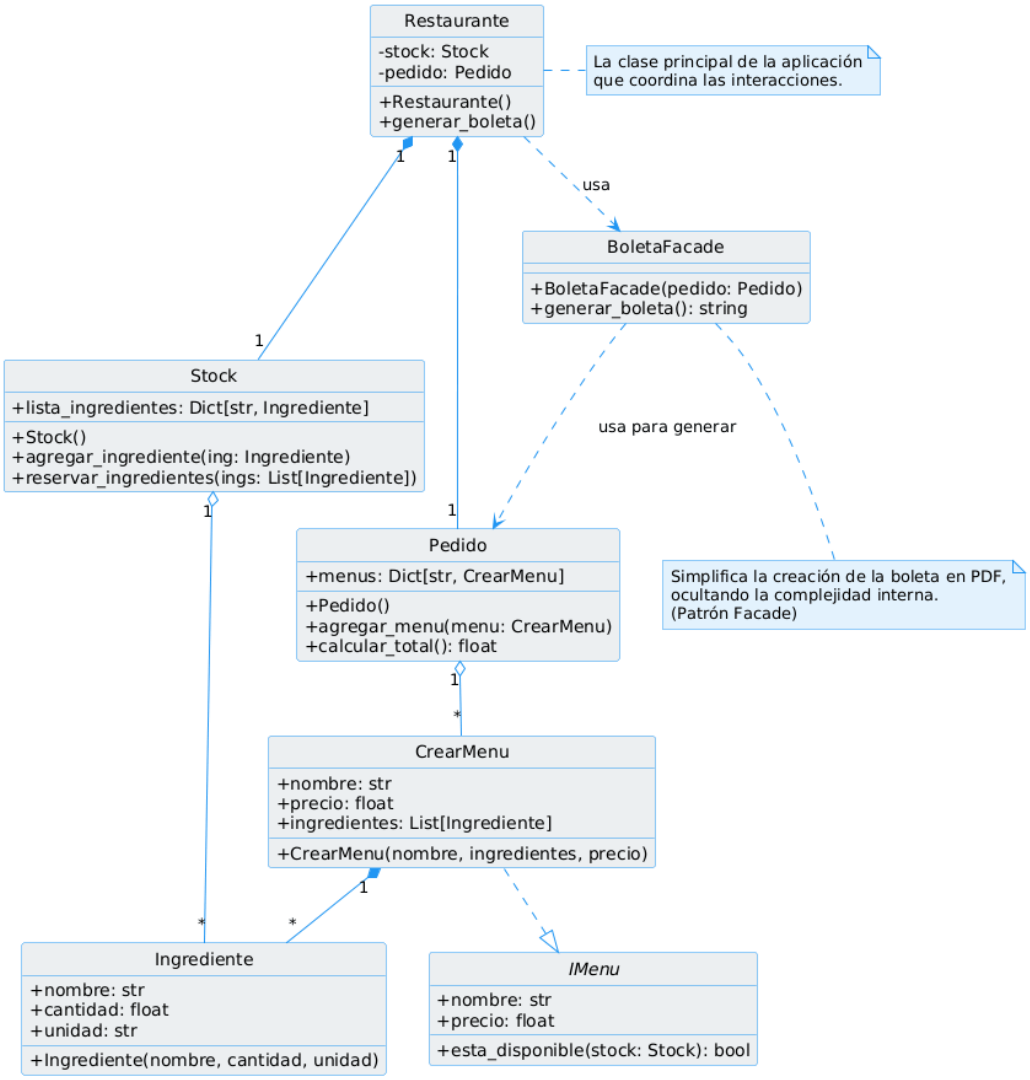

Figura 1: Diagrama de Clases del Sistema

5.2. Explicación del Diagrama

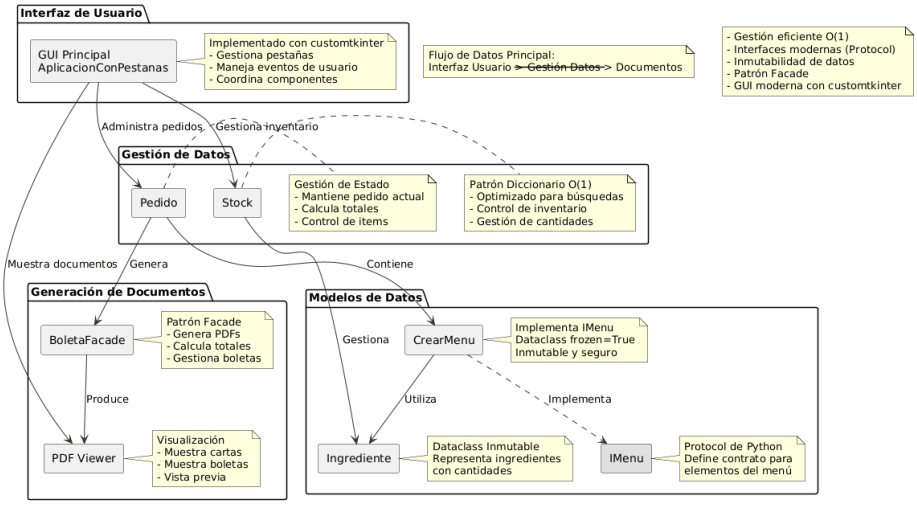

Figura 2: Explicación Detallada de las Relaciones entre Clases

5.3. Descripción de las Clases

- **AplicacionConPestanas**: Clase principal que coordina todas las funcionalidades del sistema.
- **Stock**: Gestiona el inventario de ingredientes.
- **Ingrediente**: Representa los ingredientes individuales.
- **CrearMenu**: Implementa la interfaz IMenu y representa los elementos del menú.
- **Pedido**: Maneja la gestión de pedidos.
- **BoletaFacade**: Simplifica la generación de boletas.

6. Tecnologías Utilizadas

- **Python 3.13**: Lenguaje de programación principal
- **customtkinter**: Framework para la interfaz gráfica moderna
- **FPDF**: Biblioteca para generación de PDFs
- **PyMuPDF (fitz)**: Visualización de PDFs
- **Pandas**: Procesamiento de datos CSV

7. Implementación

7.1. Gestión de Inventario

El sistema maneja el inventario a través de la clase Stock, que utiliza un diccionario (de tipo Dict[str, Ingrediente]) como estructura de datos principal. Esta decisión de diseño garantiza un rendimiento óptimo con complejidad $O(1)$ para todas las operaciones principales:

- Agregar nuevos ingredientes
- Eliminar ingredientes existentes

- Verificar disponibilidad
- Actualizar cantidades

A continuación, se muestra un ejemplo de la implementación del manejo de stock:

```python
class Stock:
    def __init__(self):
        self.lista_ingredientes: Dict[str, Ingrediente] = {}

    def agregar_ingrediente(self, ingrediente: Ingrediente):
        if ingrediente.nombre in self.lista_ingredientes:
            ing_existente = self.lista_ingredientes[ingrediente.nombre]
            nueva_cantidad = ing_existente.cantidad + ingrediente.cantidad
            ing_existente.cantidad = round(nueva_cantidad, 1)
        else:
            ingrediente.cantidad = round(ingrediente.cantidad, 1)
            self.lista_ingredientes[ingrediente.nombre] = ingrediente

    def verificar_ingredientes_suficientes(self,
            ingredientes_necesarios: List[Ingrediente]) -> bool:
        for ing_necesario in ingredientes_necesarios:
            ing_stock = self.lista_ingredientes.get(ing_necesario.nombre)
            if ing_stock is None or ing_stock.cantidad < ing_necesario.
    cantidad:
                return False
        return True
```

Listing 2: Implementación de Stock

7.2. Sistema de Pedidos

La gestión de pedidos se realiza mediante la clase Pedido, que ofrece:

- Agregar elementos al pedido
- Calcular totales
- Verificar disponibilidad de ingredientes
- Generar boletas

```python
class Pedido:
    def __init__(self):
        self.menus: Dict[str, CrearMenu] = {}

    def agregar_menu(self, menu: CrearMenu):
        if menu.nombre in self.menus:
            self.menus[menu.nombre].cantidad += menu.cantidad
        else:
            self.menus[menu.nombre] = menu

    def calcular_total(self) -> float:
        return sum(menu.precio * menu.cantidad
                   for menu in self.menus.values())
```

Listing 3: Implementación de Pedido

8. Interfaz Gráfica: Análisis por Pestaña

8.1. Configuración General de Pestañas

El sistema utiliza un sistema de pestañas implementado con customtkinter para organizar las diferentes funcionalidades:

El siguiente código muestra la inicialización del sistema de pestañas. Se utiliza un enfoque modular donde cada pestaña se configura por separado, permitiendo una mejor organización del código y facilitando el mantenimiento. La numeración de las pestañas no es secuencial por motivos de desarrollo iterativo, pero esto no afecta la funcionalidad:

```python
def crear_pestanas(self):
    # Creación de pestañas en orden lógico de uso
    self.tab3 = self.tabview.add("Carga de ingredientes")  # Primer paso: cargar ingredientes
    self.tab1 = self.tabview.add("Stock")                  # Segundo paso: verificar stock
    self.tab4 = self.tabview.add("Carta restorante")       # Tercer paso: ver carta
    self.tab2 = self.tabview.add("Pedido")                 # Cuarto paso: hacer pedido
    self.tab5 = self.tabview.add("Boleta")                 # Paso final: generar boleta

    # Configuración individual de cada pestaña
    self.configurar_pestana1()                # Configura Stock
```

```
11      self.configurar_pestana2()                  # Configura Pedido
12      self.configurar_pestana3()                  # Configura Carga de
        ingredientes
13      self._configurar_pestana_crear_menu()       # Configura Carta
14      self._configurar_pestana_ver_boleta()       # Configura Boleta
```

Listing 4: Configuración de Pestañas

8.2. Pestaña: Carga de Ingredientes

Esta pestaña permite dos métodos de ingreso de ingredientes: manual y por archivo CSV.

8.2.1. Carga Manual

La implementación de la carga manual de ingredientes incluye validaciones para asegurar la integridad de los datos. El sistema verifica que el nombre solo contenga letras y espacios, y que la cantidad sea un número válido. Además, se actualiza automáticamente la vista del inventario tras cada ingreso:

```
1  def ingresar_ingrediente(self):
2      # Obtención de datos desde la interfaz
3      nombre = self.entry_nombre.get()      # Nombre del ingrediente
4      unidad = self.combo_unidad.get()      # Unidad de medida (g, kg, l, ml
       , etc.)
5      cantidad = self.entry_cantidad.get()   # Cantidad del ingrediente
6
7      # Validaciones de datos
8      if not self.validar_nombre(nombre) or not self.validar_cantidad(
       cantidad):
9          return  # Si no pasa las validaciones, se detiene el proceso
10
11     # Creación y almacenamiento del ingrediente
12     ingrediente = Ingrediente(nombre=nombre, unidad=unidad, cantidad=float
       (cantidad))
13     self.stock.agregar_ingrediente(ingrediente)
14     self.actualizar_treeview()  # Actualización de la interfaz
```

Listing 5: Implementación de Carga Manual

8.2.2. Carga por CSV

La carga masiva de ingredientes se realiza mediante archivos CSV, lo que permite una importación eficiente de datos. El sistema utiliza pandas para el procesamiento del archivo y maneja posibles errores de forma elegante, mostrando mensajes informativos al usuario:

```
def cargar_csv(self):
    # Apertura del diálogo de selección de archivo
    filename = filedialog.askopenfilename(
        title="Seleccionar archivo CSV",
        filetypes=[("CSV files", "*.csv")]  # Solo permite archivos CSV
    )
    if filename:
        try:
            # Lectura del archivo CSV usando pandas
            df = pd.read_csv(filename)

            # Procesamiento de cada fila del archivo
            for _, row in df.iterrows():
                # Creación de objeto Ingrediente desde datos CSV
                ingrediente = Ingrediente(
                    nombre=row['nombre'],     # Nombre del ingrediente
                    unidad=row['unidad'],     # Unidad de medida
                    cantidad=float(row['cantidad'])  # Cantidad convertida
    a float
                )
                # Agregado al inventario
                self.stock.agregar_ingrediente(ingrediente)

            self.actualizar_treeview()  # Actualización de la interfaz
        except Exception as e:
            # Manejo de errores con mensaje visual
            CTkMessagebox(title="Error",
                message=f"Error al cargar el archivo: {str(e)}",
                icon="warning")
```

Listing 6: Implementación de Carga CSV

8.3. Pestaña: Stock

Muestra y gestiona el inventario actual de ingredientes.

La clase Stock implementa un sistema eficiente de gestión de inventario utilizando un diccionario como estructura de datos principal. Esta decisión de

diseño permite acceso $O(1)$ a los ingredientes y simplifica las operaciones de verificación y actualización. La clase incluye validaciones para evitar stocks negativos y manejo de ingredientes inexistentes:

```
class Stock:
    def __init__(self):
        # Diccionario para acceso O(1) a ingredientes
        self.lista_ingredientes: Dict[str, Ingrediente] = {}

    def verificar_ingredientes_suficientes(self, ingredientes: List[
    Ingrediente]) -> bool:
        """
        Verifica si hay suficiente stock para una lista de ingredientes.
        Retorna False si falta algún ingrediente o la cantidad es
    insuficiente.
        """
        for ingrediente in ingredientes:
            ing_stock = self.lista_ingredientes.get(ingrediente.nombre)
            if not ing_stock or ing_stock.cantidad < ingrediente.cantidad:
                return False  # Ingrediente no existe o cantidad
    insuficiente
        return True  # Todos los ingredientes están disponibles

    def reservar_ingredientes(self, ingredientes: List[Ingrediente]):
        """
        Descuenta las cantidades del stock para los ingredientes usados.
        Solo se llama después de verificar_ingredientes_suficientes().
        """
        for ingrediente in ingredientes:
            ing_stock = self.lista_ingredientes[ingrediente.nombre]
            ing_stock.cantidad -= ingrediente.cantidad  # Actualización at
    ómica
```

Listing 7: Gestión de Stock

8.4. Pestaña: Carta Restaurante

Permite generar y visualizar la carta del restaurante en formato PDF.

La generación de la carta en PDF combina la creación del documento con su visualización inmediata. El sistema utiliza un visor de PDF personalizado (CTkPDFViewer) que permite una experiencia integrada y fluida para el usuario:

```
def generar_y_mostrar_carta_pdf(self):
```

```
    try:
        # Configuración del archivo de salida
        pdf_path = "carta.pdf"

        # Generación del PDF con formato personalizado
        create_menu_pdf(
            self.menus,              # Lista de elementos del menú
            pdf_path,                # Ruta de salida
            titulo_negocio="Restaurante",
            subtitulo="Carta Primavera 2025",
            moneda="$"               # Símbolo monetario personalizable
        )

        # Limpieza del visor anterior si existe
        if self.pdf_viewer_carta is not None:
            self.pdf_viewer_carta.pack_forget()

        # Creación del nuevo visor con ruta absoluta
        self.pdf_viewer_carta = CTkPDFViewer(
            self.pdf_frame_carta,    # Contenedor del visor
            file=os.path.abspath(pdf_path)  # Ruta absoluta para evitar
    errores
        )
        # Configuración de expansión del visor
        self.pdf_viewer_carta.pack(expand=True, fill="both")

    except Exception as e:
        # Manejo de errores con interfaz gráfica
        CTkMessagebox(
            title="Error",
            message=f"Error al generar la carta: {str(e)}",
            icon="warning"
        )
```

Listing 8: Generación de Carta PDF

8.5. Pestaña: Pedido

Gestiona la creación y modificación de pedidos actuales.

La clase Pedido gestiona la lógica de los pedidos activos, implementando un sistema que permite acumular cantidades de menús iguales y calcular totales de forma eficiente. Utiliza un diccionario para mantener la unicidad de los menús y facilitar las actualizaciones:

```python
class Pedido:
    def __init__(self):
        # Diccionario que mapea nombres de menús a objetos CrearMenu
        self.menus: Dict[str, CrearMenu] = {}

    def agregar_menu(self, menu: CrearMenu):
        """
        Agrega un menú al pedido o incrementa su cantidad si ya existe.
        Mantiene la consistencia de datos evitando duplicados.
        """
        if menu.nombre in self.menus:
            # Si el menú ya existe, solo incrementamos la cantidad
            self.menus[menu.nombre].cantidad += menu.cantidad
        else:
            # Si es nuevo, lo agregamos al diccionario
            self.menus[menu.nombre] = menu

    def calcular_total(self) -> float:
        """
        Calcula el total del pedido usando comprensión de listas.
        Multiplica el precio unitario por la cantidad de cada menú.
        """
        return sum(menu.precio * menu.cantidad
                   for menu in self.menus.values())
```

Listing 9: Gestión de Pedidos

8.6. Pestaña: Boleta

Genera y muestra las boletas de los pedidos.

La generación de boletas se implementa utilizando el patrón Facade para simplificar la compleja tarea de crear documentos PDF. La clase BoletaFacade encapsula toda la lógica de formato, cálculos y generación del archivo, proporcionando una interfaz simple para el cliente:

```python
class BoletaFacade:
    def generar_boleta(self):
        """
        Implementación del patrón Facade para la generación de boletas.
        Coordina todos los aspectos de la creación del PDF.
        """
        # Genera los detalles y cálculos previos
        self.generar_detalle_boleta()

```

```
        # Inicialización del documento PDF
        pdf = FPDF()
        pdf.add_page()
        pdf.set_font("Arial", size=12)

        # Configuración y generación del encabezado
        pdf.set_font("Arial", 'B', 16)
        pdf.cell(0, 10, "Boleta Restaurante", ln=True, align='L')

        # Generación de la tabla de detalles
        pdf.set_font("Arial", 'B', 12)
        for item in self.pedido.menus.values():
            subtotal = item.precio * item.cantidad
            # Formato tabular con bordes
            pdf.cell(70, 10, item.nombre, border=1)        # Nombre del ítem
            pdf.cell(20, 10, str(item.cantidad), border=1) # Cantidad
            pdf.cell(35, 10, f"${item.precio:.2f}", border=1)    # Precio unitario
            pdf.cell(30, 10, f"${subtotal:.2f}", border=1)       # Subtotal
            pdf.ln()  # Nueva línea

        # Sección de totales alineada a la derecha
        pdf.cell(120, 10, "Total:", 0, 0, 'R')
        pdf.cell(30, 10, f"${self.total:.2f}", ln=True, align='R')

        # Generación del archivo con nombre único
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        pdf_filename = f"boleta_{timestamp}.pdf"  # Nombre único con timestamp
        pdf_path = os.path.join("boletas", pdf_filename)
        pdf.output(pdf_path)
        return pdf_path  # Retorna la ruta para visualización
```

Listing 10: Generación de Boletas

8.7. Componentes de la Interfaz

La interfaz utiliza varios elementos modernos de customtkinter:

8.7.1. Elementos Visuales

- Tarjetas de menú con íconos personalizados

- Visor de PDF integrado para cartas y boletas
- Tablas interactivas para gestión de datos

8.7.2. Características Avanzadas

- Validación en tiempo real de ingredientes
- Actualización automática de stock
- Previsualización de documentos PDF

8.8. Diseño Responsivo

La interfaz se adapta dinámicamente al contenido y ofrece:

- Diseño moderno con temas claro/oscuro
- Feedback visual en interacciones
- Mensajes de error y confirmación contextuales
- Organización jerárquica de información

9. Conclusiones

El sistema desarrollado cumple con los objetivos planteados, proporcionando una solución integral para la gestión de restaurantes. La implementación de patrones de diseño modernos como Protocol y principios de programación orientada a objetos permite una estructura mantenible y extensible. El uso de estructuras de datos optimizadas, como diccionarios para el manejo de inventario, asegura un rendimiento eficiente incluso con grandes volúmenes de datos.

Las decisiones de diseño tomadas, como:

- El uso de typing.Protocol para interfaces modernas
- La implementación de diccionarios para operaciones $O(1)$ en el stock
- La aplicación del patrón Facade para simplificar operaciones complejas

- La utilización de customtkinter para una interfaz gráfica moderna

Han resultado en un sistema robusto, eficiente y fácil de mantener que cumple con los requisitos del proyecto y permite futuras extensiones.

10. Anexos

10.1. Código Fuente

A continuación se presentan fragmentos relevantes del código:

```
class BoletaFacade:
    def __init__(self, pedido):
        self.pedido = pedido
        self.detalle = ""
        self.subtotal = 0
        self.iva = 0
        self.total = 0

    def generar_detalle_boleta(self):
        self.detalle = ""
        for item in self.pedido.menus:
            subtotal = item.precio * item.cantidad
            self.detalle += f"{item.nombre:<30} {item.cantidad:<10} ${item
    .precio:<10.2f} ${subtotal:<10.2f}\n"

        self.subtotal = self.pedido.calcular_total()
        self.iva = self.subtotal * 0.19
        self.total = self.subtotal + self.iva
```

Listing 11: Implementación de BoletaFacade