

Práctica 2

Ecomoda

Programación Orientada a Objetos

Grupo 01

Equipo 4

Andres David Calderón Jiménez, Gelsy Jackelin Lozano Blanquiceth, Andrea Merino Mesa,
Luis Rincon, Juanita Valentina Rosero

Jaime Alberto Guzmán Luna

Universidad Nacional de Colombia

Sede Medellín

2025

Índice

1. Descripción general de la solución.....	2
2. Descripción del diseño estático del sistema en la especificación UML.....	3
3. Descripción de la implementación de características de programación orientada a objetos en el proyecto.....	8
4. Descripción de cada una de las 5 funcionalidades implementadas.....	19
Nombre funcionalidad 1 : Gestión humana.....	19
Nombre funcionalidad 2 : Insumos.....	24
Nombre funcionalidad 3 : Sistema Financiero.....	28
Nombre funcionalidad 4 : Facturación.....	33
5. Manejo de excepciones.....	44
6. FieldFrame.....	47
7. Manual de usuario.....	50
Ventana de Bienvenida.....	50
Ventana de Inicio.....	51
Procesos y Consultas → Pedir insumos.....	55
Procesos y Consultas → Ver el desglose económico de la empresa.....	57
Procesos y Consultas → Facturación.....	60
Procesos y Consultas → Producir Prendas.....	64

1. Descripción general de la solución.

El proyecto se centra en la creación de un sistema integral que aborda los retos operativos más relevantes de una industria textil, cubriendo aspectos de gestión humana, manejo de insumos, planificación financiera, facturación y producción de prendas. Este enfoque busca no solo solucionar problemas específicos, sino también integrar las operaciones empresariales en un entorno eficiente y automatizado, optimizando el uso de recursos y minimizando errores manuales.

El análisis inicial del dominio permite identificar problemas como la dificultad para gestionar empleados de bajo rendimiento y distribuirlos de manera eficiente, la planificación inadecuada de insumos que provoca sobrecostos o desabastecimiento, la falta de herramientas para proyectar balances financieros con precisión, la inconsistencia en los precios de venta de los artículos a vender y una organización de la producción que no responde dinámicamente a las capacidades operativas. En respuesta a estos desafíos, se establecen requisitos funcionales específicos, entre ellos la capacidad de evaluar y optimizar el desempeño del personal en base a indicadores cuantitativos, realizar proyecciones de ventas basadas en datos históricos, llevar un control de los ingresos y gastos de la empresa según las deudas y ventas generadas, ajustar dinámicamente las necesidades de insumos, automatizar el proceso de calcular el subtotal de una venta, y optimizar la producción en función de la disponibilidad de máquinas y recursos humanos.

Por otro lado, se definen requisitos no funcionales que garantizan que el sistema sea confiable, eficiente y seguro. Entre estos se incluyen tiempos de respuesta rápidos para cada funcionalidad, una interfaz concisa e intuitiva para facilitar la interacción del usuario y la correcta encapsulación de los datos privados que puede tener una empresa. Además, la solución está diseñada para ser escalable, permitiendo su implementación tanto en pequeñas empresas como en organizaciones más complejas, con capacidad para adaptarse a grandes volúmenes de datos y operaciones más elaboradas.

En resumen, el proyecto representa una solución tecnológica que no solo responde a las necesidades actuales de la empresa, sino que también prevé escenarios futuros. Con su enfoque en la automatización y la optimización, el sistema busca incrementar la eficiencia, reducir costos operativos y facilitar una toma de decisiones más controlada y ágil en todas las áreas del negocio. Al implementar este sistema, la empresa estará mejor equipada para enfrentar los desafíos del mercado y mantenerse competitiva en este entorno de constante cambio.

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

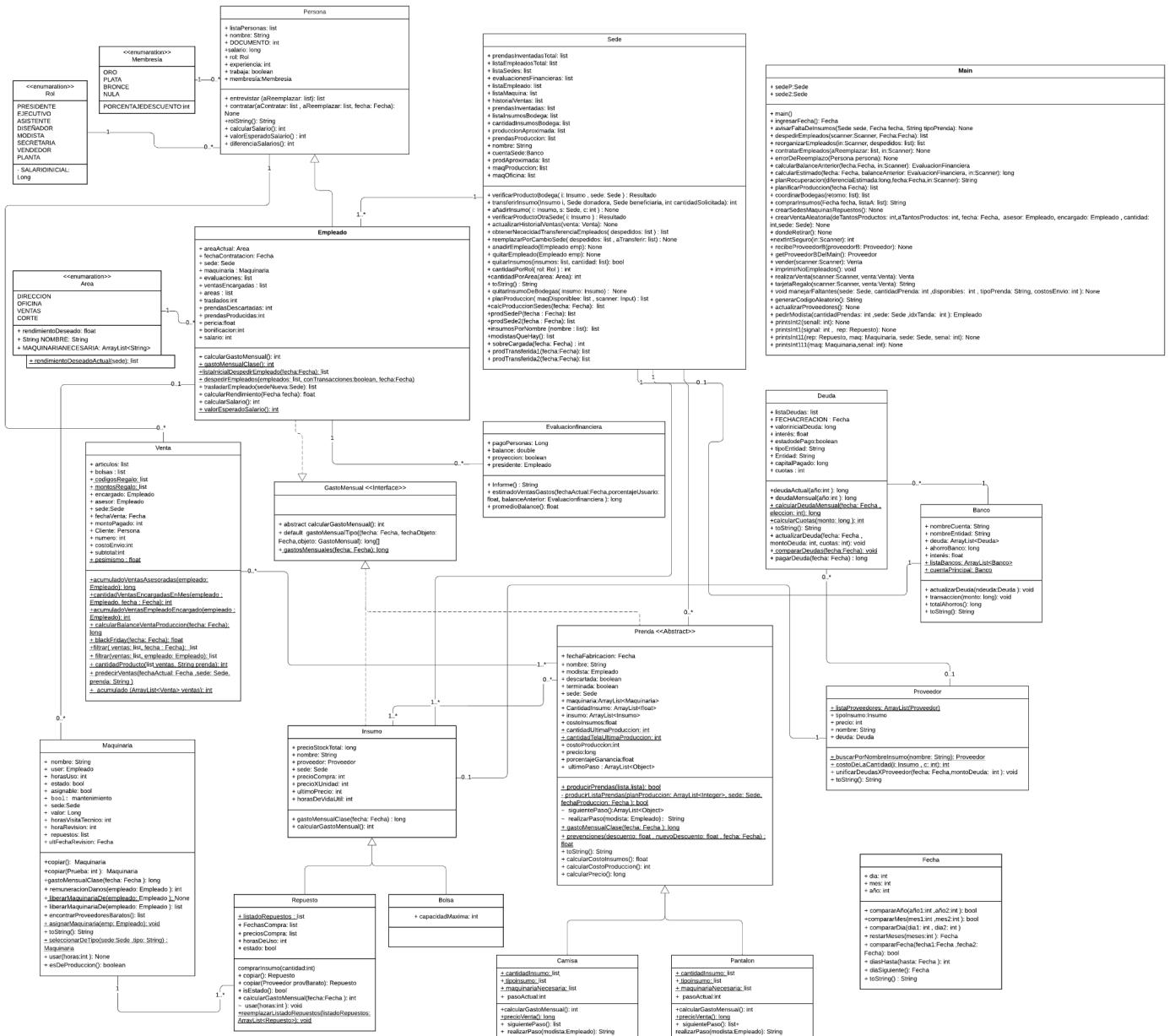


Imagen 1: diagrama de clases

Link UML: [Aquí](#)

class Main:

La clase Main actúa como el punto de entrada principal del programa. Su función es coordinar la ejecución de las funcionalidades implementadas en el sistema, invocando los métodos necesarios para inicializar y gestionar las operaciones del programa. Además, se encarga de establecer el flujo lógico de las interacciones entre las diferentes clases y módulos, garantizando así la ejecución ordenada y eficiente de cada una de las funcionalidades.

enumeration Area:

En este enumerado se definen las diferentes áreas a las que puede ser asociado un empleado, siendo la relación entre ambos de un área específica a cero o muchos empleados. Por otra parte, asigna uno o más strings que hacen referencia a los nombres de los objetos de tipo maquinaria que son asignados a los empleados de determinada área. De igual manera, tiene un método en función del cumplimiento de los empleados en el que hace un promedio de un rendimiento adecuado.

class Banco:

La clase Banco cumple la función de guardar el dinero de la empresa en diferentes cuentas con diferentes entidades bancarias, se tiene una cuenta principal que es en la que se encuentra la mayor parte del dinero y se encarga de distribuir el dinero en las demás cuentas auxiliares para que puedan cumplir con sus gastos. Tiene una relación con sede, en cuanto una sede tiene una cuenta bancaria asociada, también, desde banco se puede adquirir deudas, siendo la relación entre ambos de un banco que puede tener asociadas cero a muchas deudas.

class Deuda:

La clase Deuda se encarga de instanciar objetos de dicho tipo que sirven como un “préstamo” hacia la empresa, de una cierta cantidad de dinero, que adquiere con bancos y proveedores, estableciendo una relación con ambos de cero o muchas deudas a un solo proveedor o un solo banco. Mediante los métodos implementados puede hacer cálculos relacionados a las cuotas e intereses de cada objeto deuda en particular, así como el progreso que se ha tenido con dicha deuda.

class Empleado:

La clase Empleado hereda de persona, ambas clases comparten atributos y métodos en común y esto es útil en función de que las personas son potenciales empleados, en cuanto un empleado es una persona vinculada directamente con la empresa. Cero o muchos empleados tienen asociada un área determinada y uno o más empleados pertenecen a una sede específica. También, la clase empleado implementa la interfaz Gasto Mensual.

class EvaluacionFinanciera:

La clase EvaluacionFinanciera se encarga de instanciar objetos de este tipo, que a su vez contienen los análisis del estado financiero de la empresa, es decir la diferencia de los gastos y los ingresos. Tiene una relación con empleado en la cual cero o muchas evaluaciones financieras pueden ser atribuidas a un empleado, este concepto se usa al evaluar empleados de tipo directivo.

interface GastoMensual:

La interfaz GastoMensual se encarga de imponer el método calcularGastoMensual() a quienes heredan de ella y permite calcular los gastos, tanto fijos como variables, que tendría que cubrir la empresa cada mes, haciendo un seguimiento para seguir a flote y no quebrar.

enumeration Rol:

El enumerado Rol tiene un key que corresponde al nombre del rol y un value que es el salario inicial asociado a dicho rol, el cual puede ser modificado posteriormente. Comparte una relación con Persona, en la que un rol corresponde a cero o muchas personas.

class Bolsa:

La clase Bolsa hereda de Insumo, sirve para instanciar objetos de ese tipo que se utilizan en la parte final del proceso de facturación, tiene unos atributos que comparte con insumo y un atributo capacidadMaxima que es particular de esa clase, también sobrescribe el método de insumo getPrecioIndividual. Bolsa comparte las relaciones con proveedor y con sede al igual que su clase base Insumo.

class Camisa:

La clase Camisa hereda de la clase abstracta Prenda, por tanto sirve para instanciar objetos ya que Prenda no puede ser abstracta. Además, define los métodos siguientePaso() y realizarPaso() de Prenda para poder producir nuevas prendas en tandas según la maquinaria necesaria para producir camisas y cambiar el estado de la prenda a terminada o descartada según el caso. Camisa tiene las mismas relaciones de su clase base Prenda.

class Insumo:

Insumo es la clase que representa los materiales que necesita la empresa para la producción y venta de artículos, ya que en esta se incluyen los insumos para la producción de prendas, los repuestos de las máquinas y las bolsas de compra. Por ello, esta clase permite

llevar un control de los insumos que tiene actualmente la empresa para saber si se pueden usar o si hay que surtir. Insumo implementa la interfaz Gasto Mensual y además tiene relaciones con proveedor y sede, de cero a un insumo tiene un proveedor y de un insumo a muchos pueden corresponder a una sede.

class Maquinaria:

La clase Maquinaria sirve para instanciar los objetos que representan las máquinas de nuestra empresa, todas ellas son operadas por algún empleado según su área y contienen diferentes repuestos que deben cambiarse cada cierto tiempo de uso. Dicho uso se cuenta al producir prendas ya que estas se van desgastando y necesitan revisión al acumular ciertas horas de uso. Asimismo, cero o muchas maquinarias pueden tener cero a un empleado encargado de ellas y una maquinaria está asociada con uno a muchos repuestos.

class Pantalon:

La clase Pantalon hereda de la clase abstracta Prenda, por tanto sirve para instanciar objetos ya que Prenda no puede ser abstracta. Además, define los métodos siguientePaso() y realizarPaso() de Prenda para poder producir nuevas prendas en tandas según la maquinaria necesaria para producir pantalones y cambiar el estado de la prenda a terminada o descartada según el caso. Pantalón tiene las mismas relaciones de su clase base Prenda.

abstract class Prenda:

La clase abstracta Prenda agrupa las características y elementos comunes de Camisa y Pantalón, haciendo que ellas puedan ser referenciadas en múltiples cálculos y listas a pesar de ser diferentes. Además, define métodos abstractos para que sus subclases, al ser diferentes, puedan adaptarlo a su necesidad como es el caso de siguientePaso() y realizarPaso(). Por otro lado, al ser abstracta y no permitir instanciarse permite que haya coherencia en el código pues evita que se creen prendas sin definir un tipo específico. Prenda implementa la interfaz Gasto mensual, además, cuenta con relaciones como: una Prenda tiene asociada una fecha, un empleado, una o muchas maquinarias, uno o muchos insumos y muchas prendas tienen una sede.

class Proveedor:

La clase proveedor es la que permite comprar insumos, pues cada proveedor tiene un insumo y un precio que le asigna a este, por lo que se pueden comparar entre ellos, escoger y comprar, lo que permite a su vez generar deudas según la cantidad comprada y el precio por unidad. Un proveedor puede tener de cero deudas a muchas y puede tener de cero insumos a uno.

class Repuesto:

Esta clase hereda de insumo y representa una pieza de la maquinaria que debe ser cambiada cada cierto tiempo, por lo que permite comparar las horas de cambio del insumo con las actuales de uso y en caso tal, reponer dicho repuesto para que la máquina siga funcionando. Además de las relaciones que hereda de la clase base Insumo, cada repuesto cuenta con una fecha, y de uno a muchos repuestos corresponden a una maquinaria.

class Fecha:

Esta clase permite manejar un formato de Fecha en el programa donde se tiene más control del tiempo actual ya que el usuario puede ingresar la fecha deseada y por tanto de puede simular el paso del tiempo sin tener que afectar el funcionamiento del programa.

enum Membresia:

El enumerado membresía tiene un key que corresponde al nombre de la membresía y un value que es el descuento asociado a dicha membresía, el cual es usado para calcular descuentos cuando una persona quiere hacer una compra y a la vez hacer un estimado de las ventas a futuro sabiendo qué, en la jerarquía de membresías (oro, plata, bronce, nula) se estima que las personas que tienen mayor membresía tienen más probabilidades de seguir comprando artículos.

class Persona:

La clase persona es la que contiene a todas las personas de forma indistinta, no importa si son o no empleados. Esto nos permite hacer comparaciones entre los objetos que también son empleados y los que no. Sin embargo, también permite realizar otras funciones como venderle a un cliente, que será de tipo Persona.

class Sede:

La clase sede lleva el registro de lo que se tiene en cada una de las instancias, lo que nos permite rastrear datos, verificar cantidades, hacer traslados, transferencias y traslados con los objetos de tipo empleado, insumo y prenda que existen en cada sede.

class Venta:

La clase venta es la que permite generar nuevas ventas y almacenar el historial de estas, puesto que este dato luego se usa para calcular cuánto se debería producir, qué tanto están vendiendo los empleados de ciertas áreas, qué tantas ventas se esperan para el próximo

mes, entre otros datos que ayudan a guiar los planes a futuro.

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

Clases Abstractas

La clase abstracta Prenda es muy importante en el diseño del programa, el uso de la abstracción evita la creación de objetos genéricos innecesarios y garantiza que solo se trabajen instancias específicas, mejorando la organización del código y reduciendo posibles errores. Al definir características y comportamientos comunes que son heredados por las subclases Pantalón y Camisa, se promueve la reutilización del código y se facilita la expansión del sistema al añadir nuevos tipos de prendas en el futuro.

```
class Prenda(ABC, GastoMensual):
    porcentajeGanancia = 0.40
    cantidadUltimaProduccion = 0
    cantidadTelaUltimaProduccion = 0
    sobreCostoPorTrabajoExtra=0

    def __init__(self, fecha: Fecha, sede: Sede, nombre: str, modista: Empleado,
                self.fechaFabricacion = fecha
                self.sede = sede
                sede.prendasInventadas.append(self)
                sede.getPrendasInventadasTotal().append(self)
                self.nombre = nombre
                self.modista = modista
                self.sobreCostoPorTrabajoExtra = 0
                self.cantidadUltimaProduccion = 0
                self.cantidadTelaUltimaProduccion = 0
```

Imagen 2: clase abstracta Prenda, archivo Bodega, paquete gestorAplicacion

Método Abstracto siguientePaso

El método abstracto siguientePaso permite que cada prenda defina este método de forma particular y acorde a sus necesidades específicas, asegurando flexibilidad y personalización en el comportamiento del sistema. Esto es especialmente útil en el proyecto, ya que respalda la extensibilidad del código, facilita la adición de nuevos tipos de prendas con comportamientos únicos, y fomenta un diseño robusto mediante el uso de polimorfismo.

```
@abstractmethod
def siguientePaso(self):
    pass
```

Imagen 3: declaración del método siguientePaso en Prenda en la línea 118 de Prenda

```

def siguientePaso(self) -> List[Optional[int]]:
    retorno = []
    if self.pasoActual == 1:
        retorno.extend(["Maquina de Corte", 5])
    elif self.pasoActual == 2:
        retorno.extend(["Maquina de Coser Industrial", 10])
    elif self.pasoActual == 3:
        retorno.extend(["Plancha Industrial", 5])
    elif self.pasoActual == 4:
        retorno.extend(["Bordadora Industrial", 5])
    elif self.pasoActual == 5:
        retorno.extend(["Maquina de Termofijado", 10])
    else:
        retorno.append("LISTO")
    self.ultimoPaso = retorno
    return retorno

```

Imagen 4: implementación del método `siguientePaso` en camisa desde la línea 40

```

def siguientePaso(self):
    retorno = []
    if self.pasoActual == 1:
        retorno.append("Maquina de Corte")
        retorno.append(5)
    elif self.pasoActual == 2:
        retorno.append("Maquina de Tijereado")
        retorno.append(2)
    elif self.pasoActual == 3:
        retorno.append("Maquina de Coser Industrial")
        retorno.append(10)
    else:
        retorno.append("LISTO")
        self.terminada = True
    self.ultimoPaso = retorno
    return retorno

```

Imagen 5: implementación del método `siguientePaso` en pantalón desde la línea 33.

Interfaces

La interfaz GastoMensual desempeña un papel clave en el programa al definir un marco común para calcular los gastos mensuales de la empresa. Todas las clases que generan gastos implementan esta interfaz, asegurando que cada una ofrezca su propia lógica para calcular su contribución a los egresos totales. Luego, un método específico se encarga de consolidar todos los gastos mensuales implementados en las clases y se suman los egresos, proporcionando una visión completa de los costos de la empresa. Esto no solo mejora la organización y la cohesión del código, sino que también facilita el mantenimiento y la

extensibilidad del sistema al permitir agregar nuevas fuentes de gastos sin alterar el flujo principal del programa.

```
class GastoMensual(ABC):
    @abstractmethod
    def calcularGastoMensual(self):
        pass
```

Imagen 6: clase GastoMensual, archivo Administracion, paquete gestorAplicacion

Método por default

El método por defecto gastoMensualTipo, definido en la interfaz GastoMensual, permite a las clases que la implementan filtrar y calcular los gastos según la fecha actual, determinando los gastos correspondientes al mes actual y al mes inmediatamente anterior, en caso de que existan gastos en el mes en curso. Este diseño facilita la coherencia en la lógica de filtrado entre las diferentes clases que manejan gastos y asegura que el sistema pueda adaptarse dinámicamente a escenarios en los que no haya registros de gastos para el mes actual, ofreciendo un manejo confiable de datos históricos. Gracias a su implementación por defecto, las clases pueden aprovechar esta funcionalidad común sin necesidad de redefinirla, promoviendo la reutilización del código y simplificando el desarrollo y mantenimiento del sistema.

```
def gastoMensualTipo(self, fecha, fechaObjeto, objeto):
    gastoActual = 0
    gastoPasado = 0
    gastoTotal = [0, 0]
    if fechaObjeto.getAno() == fecha.getAno():
        if fechaObjeto.getMes() == fecha.getMes():
            gastoActual += objeto.calcularGastoMensual()
            gastoTotal[0] = gastoActual
        if fechaObjeto.getMes() == fecha.getMes() - 1:
            gastoPasado += objeto.calcularGastoMensual()
            gastoTotal[1] = gastoPasado
    return gastoTotal
```

Imagen 7: método por default gastoMensualTipo a partir de la línea 9 en la clase GastoMensual

Método estático

El método estático gastosMensuales de la clase GastoMensual centraliza el cálculo de los gastos mensuales de la empresa. Este método se encarga de invocar las clases responsables de generar gastos, como insumos o proveedores, y recopilar los gastos

correspondientes al mes en curso. Luego, suma todos estos valores para obtener el total de egresos, excluyendo los gastos provenientes de las clases no forman parte directamente de la estructura financiera de la empresa, como por ejemplo Banco. Este enfoque permite una gestión eficiente de los gastos sin necesidad de recorrer cada clase individualmente de manera manual. Al ser un método estático, simplifica la llamada desde cualquier parte del programa, garantizando que todos los gastos se calculen de manera coherente y consolidada, mejorando la organización y eficiencia del código.

```
@staticmethod  
def gastosMensuales(fecha):  
    from src.gestorAplicacion.administracion.empleado import Empleado  
    from src.gestorAplicacion.bodega.insumo import Insumo  
    from src.gestorAplicacion.bodega.maquinaria import Maquinaria  
    gastosMaquinaria = Maquinaria.gastoMensualClase(fecha)  
    gastosNomina = Empleado.gastoMensualClase()  
    gastoBolsa = Insumo.gastoMensualClase(fecha)  
    suma = gastosMaquinaria + gastosNomina + gastoBolsa  
    return suma
```

Imagen 8: método estático `gastosMensuales` a partir de la línea 22 en la clase `GastoMensual`

Herencia

La clase Empleado que hereda de Persona es una implementación clave en el diseño del programa, ya que permite que la clase Empleado herede atributos comunes como nombre, documento, rol, entre otros, de la clase Persona. Esta relación facilita la reutilización de código y asegura que tanto Empleado como Persona comparten características comunes, lo que optimiza la gestión de datos relacionados con las personas en el sistema. La ventaja de esta implementación es que cualquier persona puede ser considerada un Empleado en el contexto del sistema, simplemente cuando se encuentre vinculada o contratada por la empresa.

```
class Persona:  
    listaPersonas = []  
  
    def __init__(self, nombre: str, documento: int, rol: Rol, experiencia: int,  
                 self.nombre = nombre  
                 self.documento = documento  
                 self.rol = rol  
                 self.experiencia = experiencia  
                 self.trabaja = trabaja  
                 self.membresia = membresia  
                 self.salario = self.calcularSalario()  
                 Persona.listaPersonas.append(self)
```

Imagen 9: clase `Persona`, paquete `gestorAplicacion`

```

class Empleado(Persona, GastoMensual):
    @multimethod
    def __init__(self, areaActual: Area, fecha: Fecha, sede: Sede, nombre: str, documento: int, rol: Rol, experiencia: Persona):
        self.bonificacion = 0
        Persona.__init__(self, nombre, documento, rol, experiencia, True, membresia)

```

Imagen 10: clase Empleado extendida de Persona, archivo Administracion, paquete gestorAplicacion

Ligadura dinámica

En general, en Python la herencia se resuelve por ligadura dinámica, sin embargo, en el programa se presentan dos casos muy interesantes donde se puede apreciar particularmente como esto sucede.

En el método getPrecioIndividual de la clase Insumo, el cual es heredado por la clase Bolsa. En el proceso de iteración sobre los insumos en bodega, por cada objeto Insumo, se llama al método getPrecioIndividual que devuelve el precio del insumo y lo multiplica por la cantidad presente en la bodega. Sin embargo, cuando se encuentra con un objeto de tipo Bolsa, se ejecuta la versión sobrescrita de getPrecioIndividual que maneja el cálculo del precio de manera especial, ya que una bolsa puede tener un descuento según el proveedor. Gracias a la ligadura dinámica, el método adecuado es llamado según el tipo real del objeto en tiempo de ejecución, permitiendo que la iteración se maneje de manera flexible y eficiente, sin necesidad de conocer de antemano si se trata de un Insumo o una Bolsa.

```

        self.precioCompra - precio
def getPrecioIndividual(self):
    return self.precioXUnidad
def __init__(self, nombre, documento, rol, experiencia, True, membresia):

```

Imagen 11: método getPrecioIndividual en la línea 69 de la clase Insumo

```

def getPrecioIndividual(self):
    return round(self.precioXUnidad - (self.precioXUnidad * self.proveedor.getDescuento() * self.capacidadMaxima))

```

Imagen 12: método getPrecioIndividual en la línea 9 de la clase Bolsa

```

def calcularGastoMensual(self):
    valor = 0
    for i in range(len(self.sede.getListaNsumosBodega())):
        if self.sede.getListaNsumosBodega()[i] == self:
            valor = self.getPrecioIndividual() * self.sede.getCantidadInsumosBodega()[i]
    return valor

```

Imagen 13: implementación de la ligadura dinámica en el método calcularGastoMensual desde la línea 40 de la clase Insumo

De igual manera, en el método diferenciarSalarios explicado anteriormente, que llama al método valorEsperadoSalario, que a su vez llama al método calcularSalario cuyo

comportamiento varía dinámicamente dependiendo del tipo del objeto en tiempo de ejecución, así: para las instancias de Persona, el método considera únicamente la experiencia como factor determinante en el cálculo del salario, en cambio, para las instancias de Empleado, se utiliza una implementación más específica que incluye elementos adicionales, como las cesantías y las bonificaciones asociadas a cada empleado. Este uso de ligadura dinámica asegura que cada clase puede implementar el cálculo del salario según sus propios requerimientos particulares

```
def calcularSalario(self) -> int:  
    return round((self.rol.getSalarioInicial() * 0.05) * self.experiencia) + self.rol.getSalarioInicial()
```

Imagen 14: método calcularSalario en la línea 83 de la clase Persona

```
def calcularSalario(self):  
    return Persona.calcularSalario(self) + self.getBonificacion()
```

Imagen 15: método calcularSalario en la línea 52 de la clase Empleado

Atributo de clase

En la clase Venta se puede verificar que existe un atributo de clase, denominado pesimismo, este atributo representa un porcentaje que refleja posibles factores negativos que podrían afectar las ventas. El atributo pesimismo es particularmente útil porque permite a todas las instancias de Venta acceder a una misma referencia global sobre el nivel de riesgo o incertidumbre económica, sin necesidad de que cada instancia almacene esta información por separado. Este diseño mejora la eficiencia del programa y facilita el ajuste dinámico del sistema ante posibles cambios en el entorno económico, ya que modificar el valor de pesimismo afecta de manera inmediata a todas las ventas. Además, este atributo contribuye a realizar cálculos más realistas y a tomar decisiones fundamentadas al momento de proyectar ingresos o analizar el impacto de factores externos en la empresa.

```
class Venta:  
    codigosRegalo = []  
    montosRegalo = []  
    pesimismo = 0.02
```

Imagen 16: atributo de clase pesimismo en la línea 12 de la clase Venta, paquete gestorAplicacion

Método de clase

El método de clase predecirVentas permite realizar una proyección de las ventas totales de la empresa sin necesidad de depender de las instancias específicas de Venta. Este método es llamado directamente desde la clase y utiliza datos compartidos o cálculos basados

en parámetros generales, como datos históricos, para generar una predicción global. La utilidad de este método radica en su capacidad para proporcionar una visión general de las posibles ventas futuras, lo cual es invaluable para la planificación estratégica y la toma de decisiones de alto nivel. Al no depender de instancias específicas, predecirVentas se convierte en una herramienta eficiente y centralizada para calcular tendencias y ajustar estrategias de negocio en función de las proyecciones generadas.

```
@classmethod
def predecirVentas(cls, fechaActual, sede, prenda):
    from src.gestorAplicacion.sede import Sede
    ventasMes1 = cls.cantidadProducto(cls.filtrar(Sede.getHistorialVentas(sede), Fecha.restarMeses(fechaActual, 3)),
    ventasMes2 = cls.cantidadProducto(cls.filtrar(Sede.getHistorialVentas(sede), Fecha.restarMeses(fechaActual, 2)),
    pendienteMes1a2 = ventasMes2 - ventasMes1
    ventasMes3 = cls.cantidadProducto(cls.filtrar(Sede.getHistorialVentas(sede), Fecha.restarMeses(fechaActual, 1)),
    pendienteMes2a3 = ventasMes3 - ventasMes2
    pendientePromedio = (pendienteMes1a2 + pendienteMes2a3) / 2
    return math.ceil(ventasMes3 + pendientePromedio)
```

Imagen 17: método de clase predecirVentas en la línea 123 de la clase Venta, paquete gestorAplicacion

Uso de una constante

En Python, no existe la palabra clave “final” como en Java para definir constantes, pero se sigue una convención bastante clara en la que se nombra el atributo utilizando mayúsculas, lo cual indica que no debe cambiarse, como sucede con el atributo DOCUMENTO de la clase Persona. Esta característica tiene un propósito claro en el contexto del programa, ya que el número de documento de una persona es un identificador único y fijo, que no cambia a lo largo de su vida. El uso de dicha constante garantiza la integridad y consistencia de la información.

```
def __init__(self, nombre: str, documento: int,
            self.nombre = nombre
            self.DOCUMENTO = documento
            self.rol = rol
            self.experiencia = experiencia
            self.trabaja = trabaja
            self.membresia = membresia
            self.salario = self.calcularSalario()
            Persona.listaPersonas.append(self)
```

Imagen 18: atributo final documento en la línea 11 de la clase Persona, paquete gestorAplicacion

Encapsulamiento

En Python todo es público, sin embargo por convención y buenas prácticas, se accede a los atributos de cada clase a través de getters y setters.

```

def getDia(self):
    return self.dia
def setDia(self, dia):
    self.dia = dia
def getMes(self):
    return self.mes
def setMes(self, mes):
    self.mes = mes
def getAno(self):
    return self.ano
def setAno(self, ano):
    self.ano = ano

```

Imagen 19: se acceden a los atributos de la clase Fecha a través de métodos get y set

Sobrecarga de métodos

La sobrecarga de métodos en Python funciona de una manera un poco diferente a otros lenguajes como Java, porque Python no soporta sobrecarga tradicional basada en el tipo o número de argumentos. Sin embargo, hay formas de lograr comportamientos similares, como con multimethod, lo cual se usa para la sobrecarga del método filtrar en la clase Venta, este es un ejemplo clave de cómo la misma función puede adaptarse a diferentes necesidades, dependiendo de los parámetros que se pasen. Este concepto es fundamental en la programación orientada a objetos, ya que permite que un solo nombre de método ejecute múltiples comportamientos según el contexto en que se llame.

En el caso de Venta, la sobrecarga se utiliza para realizar dos procesos ligeramente diferentes, pero relacionados, dependiendo de los argumentos proporcionados al método. Por ejemplo, el método puede filtrar ventas por diferentes criterios como fecha, tipo de producto, o monto total, utilizando una versión u otra del método.

```

@multimethod
def filtrar(cls,ventas>List, fecha:Fecha):
    ventasMes = []
    for venta in ventas:
        if venta.fechaVenta.ano == fecha.ano and venta.fechaVenta.mes == fecha.mes:
            ventasMes.append(venta)
    return ventasMes
from src.gestorAplicacion.administracion.empleado import Empleado

```

Imagen 20: método filtrar en la clase Venta, paquete gestorAplicacion, desde la línea 96

```

@mymethod
def filtrar(cls, ventas:List, empleado:Empleado):
    asesoradas = []
    for venta in ventas:
        if venta.asesor == empleado:
            asesoradas.append(venta)
    return asesoradas
filtrar = classmethod(filtrar)

```

Imagen 21: método filtrar sobre cargado en la clase Venta, paquete gestorAplicacion, desde la línea 105

Primer caso de sobrecarga de constructores

La sobrecarga de constructores se maneja con la librería multimethod de manera que se define el tipo de los valores para diferenciar entre los parámetros necesarios para los diferentes constructores, es así como en la clase Empleado se facilita la creación de instancias del objeto Empleado a partir de diferentes conjuntos de argumentos. Este concepto es muy útil en situaciones donde no todos los atributos de un objeto deben ser proporcionados al momento de la creación, permitiendo que el objeto se inicialice de manera flexible con solo los parámetros necesarios en cada caso.

```

@mymethod
def __init__(self, areaActual: Area, fecha: Fecha, sede: Sede, nombre: str, documento: int, rol: Rol, experiencia: int, membresia: Membresia=Membresia):
    self.bonificacion = 0
    Persona.__init__(self, nombre, documento, rol, experiencia, True, membresia)
    self.areaActual = areaActual
    self.traslados = 0
    self.areas = [] # Areas por las que ha pasado
    self.sede = sede
    self.maquinaria = maquinaria
    self.fechaContratacion = fecha
    self.prendasDescartadas = 0
    self.prendasProducidas = 0
    self.pericia = random.uniform(0.9,1)
    self.evaluciones = []; self.ventasEncargadas = []
    sede.anadirEmpleado(self)
    Sede.getListaEmpleadosTotal().append(self)

@mymethod
def __init__(self, area: Area, fecha: Fecha, sede:Sede, p: Persona):
    self.traslados = 0
    self.bonificacion=0
    self.prendasDescartadas = 0
    self.prendasProducidas = 0
    self.maquinaria = None
    Persona.__init__(self,p.getNombre(),p.getDocumento(),p.getRol(),p.getExperiencia(),p.isTrabaja(),p.getMembresia())
    Sede.getListaEmpleadosTotal().append(self)
    self.areaActual = area
    self.evaluciones = []; self.ventasEncargadas = []; self.areas= []
    self.fechaContratacion = fecha
    self.sede = sede
    self.pericia = random.uniform(0.9,1)
    sede.anadirEmpleado(self)

```

Imagen 22: sobrecarga del constructor de la clase Empleado desde la línea 15, archivo Administracion, paquete gestorAplicacion,

Segundo caso de sobrecarga de constructores

La sobrecarga de constructores se maneja con la librería multimethod, permitiendo definir distintos métodos `__init__` diferenciados por el tipo de sus parámetros. En este caso, se implementan dos versiones del constructor: una que recibe un objeto de tipo Maquinaria y otra que acepta valores individuales como nombre, valor, horaRevision, repuestos y sede. Esta implementación facilita la creación de instancias de la clase a partir de diferentes conjuntos de datos, brindando flexibilidad en la inicialización del objeto. Este enfoque es útil en escenarios donde los atributos pueden provenir de distintas fuentes, el primer caso se usa para crear instancias a partir de una maquinaria ya existente y el segundo caso crea la instancia de cero.

```
@multimethod
def __init__(self, maquina: 'Maquinaria', repuestos: list):
    self.nombre = maquina.nombre
    self.sede = maquina.sede
    self.valor = maquina.valor
    self.horaRevision = maquina.horaRevision
    self.repuestos = repuestos
    self.sede.getListaMaquinas().append(self)
    self.asignarRepAsedes(self.sede, repuestos)
    self.horasUso = 0
    self.user = None
    self.estado = True
    self.asignable = True
    self.mantenimiento = False
    self.horasVisitaTecnico = 0
    self.ultFechaRevision = None

@multimethod
def __init__(self, nombre: str, valor: int, horaRevision: int, repuestos: list, sede: Sede):
    self.nombre = nombre
    self.user = None
    self.horasUso = 0
    self.estado = True
    self.asignable = True
    self.mantenimiento = False
    self.sede = sede
    self.valor = valor
    self.horasVisitaTecnico = 0
    self.horaRevision = horaRevision
    self.repuestos = repuestos
    sede.getListaMaquinas().append(self)
    self.asignarRepAsedes(sede, repuestos)
    self.ultFechaRevision = None
```

Imagen 23: sobrecarga del constructor de la clase Maquinaria desde la línea 5, archivo Bodega, paquete gestorAplicacion

Primer caso de self.

Cuando un constructor recibe parámetros con nombres idénticos a los atributos de la clase, el uso de `self` permite que el programa distinga claramente entre los parámetros locales y los atributos de la instancia, el uso de `this` facilita la asignación correcta de valores, haciendo que el proceso de inicialización del objeto sea más eficiente y preciso.

```

class Maquinaria:
    def __init__(self, nombre: str, valor: int, horaRevision: int,
                 horasUso: int):
        self.nombre = nombre
        self.user = None
        self.horasUso = horasUso
        self.estado = True
        self.asignable = True

```

Imagen 24: uso de self. desde la línea 7 del constructor de Maquinaria, archivo Bodega, en el paquete gestorAplicacion

Segundo caso de self.

El método calcularPrecio hace uso de la palabra clave this para referirse a los atributos de la clase en la que se encuentra. Esto es necesario cuando el método necesita acceder a los atributos de la instancia actual de la clase, pero hay una posible confusión con variables locales o parámetros con el mismo nombre.

El uso de self garantiza que el método esté accediendo correctamente a los atributos de la instancia (como el precio base o los descuentos), en lugar de intentar utilizar variables locales o parámetros que podrían tener el mismo nombre pero diferente ámbito.

```

def calcularPrecio(self):
    costoTotal = self.costoInsumos + self.costoProduccion
    gananciaDeseada = costoTotal + (costoTotal * Prenda.porcentajeGanancia)
    self.precio = round(gananciaDeseada)
    return self.precio

```

Imagen 25: uso de self. en la línea 170 y 173 del método calcularPecio de Prenda, archivo Bodega, en el paquete gestorAplicacion

Casos de this()

En python no existe el this () para llamar métodos sobrecargados, si se desea llamar a otro constructor dentro de la misma clase, se debe usar super() si hay herencia, o llamar explícitamente a self.__init__() si se está haciendo sobrecarga de métodos.

Sin embargo, al estar usando multimethod, este mecanismo de sobrecarga impide que self.__init__() llame a otro método sobrecargado dentro de la misma clase.

Caso de enumeración

El enumerado Membresía en el proyecto es fundamental para gestionar los diferentes

niveles de membresía que los clientes pueden tener. Al usar un enum, el código se vuelve más claro y semánticamente correcto, ya que cada tipo de membresía: ORO, PLATA, BRONCE o NULA, se representa con un valor constante, lo cual es útil en situaciones donde se sabe que solo pueden existir ciertos valores predefinidos.

```
class Membresia(Enum):
    ORO=0.30; PLATA=0.15; BRONCE=0.5; NULA=0.0
    def __init__(self, descuento):
        self.porcentajeDescuento = descuento
    def getPorcentajeDescuento(self):
        return self.porcentajeDescuento
```

Imagen 26: enumerado Membresia, paquete gestorAplicacion

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

Nombre funcionalidad 1 : Gestión humana

Interacción 1

En la primera interacción el método *despedirEmpleados* obtiene una lista de empleados que no rinden correctamente utilizando la *listaInicialDespedirEmpleado* que, en todas las sedes, va por cada empleado y mide su rendimiento de la siguiente manera, dependiendo de su área.

- Para el área de ventas, es el monto promedio de sus propias ventas.
- Para los de el área oficina, es la proporción de 0 a 100 de las ventas en las que estuvieron de encargados y el promedio de la sede.
- Para dirección, es el porcentaje de balances positivos entre los totales.
 - Las evaluaciones financieras son asociadas previamente a los empleados de dirección, en forma de datos por defecto o en gestión financiera.
 - Las calificaciones negativas que no sean pérdidas mayores al 20% del promedio de los balances tienen 50% menos peso en el cálculo.
- Para corte, es la proporción, de 0 a 100 entre prendas completadas y prendas producidas.

Cada área por cada sede tiene un rendimiento deseado

- Si es de dirección, es del 60%
- Si es de oficina, promedio de ventas por empleado en el mes.
- Si es de ventas, es el 80% del promedio de montos totales de ventas de los empleados de la sede en el mes.
- Si es de corte, el rendimiento deseado es el 90% de la proporción promedio entre prendas arruinadas y producidas por cada empleado.

Cuando el rendimiento del empleado sea menor al rendimiento deseado, se añade a una lista, llamada la lista a despedir.

En caso de que el empleado no sea del área de corte y tenga menos de 2 traslados se puede sacar de la lista y llevarlo a un área menor.

Es además posible transferir a un empleado a otra sede si el mismo área en otra sede tiene un rendimiento deseado que no sea mayor por 20 o más al rendimiento del empleado. En ese caso, se le quita de la lista a despedir.

Se muestra en la ventana inicial la diferencia salarial promedio de sus empleados y todas las personas, junto con la lista de empleados a despedir. El usuario puede modificar esta lista guía para añadir a gente que quiera despedir, y posteriormente el usuario selecciona de la lista los empleados que debe despedir.

La capa lógica se encarga de despedir la selección de empleados a través del método *despedirEmpleados*, también elimina al empleado de las listas, paga cesantías y registra el pago a la empresa por daños a las máquinas.

```
# Metodos asistentes a la versión grafica de la interacción 1.

@classmethod
def listaInicialDespedirEmpleado(cls):
    return Empleado.listaInicialDespedirEmpleado(Main.fecha)

@classmethod
def despedirEmpleados(cls, nombres):
    empleados = []
    for nombre in nombres:
        encontrado=False
        for emp in Sede.getListaEmpleadosTotal():
            if emp.getNombre().lower() == nombre.lower():
                empleados.append(emp)
                encontrado=True
        if not encontrado:
            return (False,[])
    Empleado.despedirEmpleados(empleados, True, Main.fecha)
    cls.despedidos = empleados
    cls.porReemplazar = empleados.copy()
    return (True,empleados)

@classmethod
def verificarSedeExiste(cls, sede:str): # verifica que la sede exista
    return Sede.sedeExiste(sede)
```

Interacción 2

Ahora a reemplazar los empleados despedidos antes por medios diferentes a la contratación. La lista de despedidos es ahora la lista de empleados a reemplazar, esta interacción sucede en el método *prepararCambioSede* de Main.

La primera opción es transferir empleados de otras sedes, se encarga entonces a la capa lógica por el método de sede *obtenerNececidadTransferenciaEmpleados* de juzgar que tan “sobretrabajado” está cada rol en cada sede (solo aplica para modistas y secretarias), con los siguientes criterios:

- En el caso de los modistas, debe haber menos de 30 producidas por modista por mes en la sede donadora para que pueda donar.
- En el caso de secretarias, debe haber menos de 18 empleados y menos de 2 ejecutivos por secretaria para poder donar.

Se elimina de la lista a reemplazar los empleados cuyo rol pueda ser suplido con el personal de otras sedes y se da a main una lista nueva de empleados a reemplazar.

Se muestra en pantalla de cada rol empleados que se pueden transferir de la sede donante y se le brinda información relevante para ello, como las ventas asesoradas para vendedores, la pericia para modistas y la fecha de contratación para los demás.

Teniendo la selección del usuario, se encarga a la capa lógica que reemplace a los despedidos usando los empleados seleccionados para transferir por el usuario, esto sucede en *reemplazarPorCambioSede*.

Va a quedar una lista de empleados que no se pudieron suplir de esta manera, esto pasa a la tercera interacción.

```
# Ejecutado al pasar a la interacción 2 grafica.
@classmethod
def prepararCambioSede(cls):
    nececidad = Sede.obtenerNecesidadTransferenciaEmpleados(Main.despedidos)
    cls.rolesAReemplazar = nececidad[0] if nececidad else []
    cls.transferirDe = nececidad[1] if nececidad else []
    cls.aContratar = nececidad[2] if nececidad else []
    cls.seleccion = []
    cls.idxRol = 0
    return cls.getTandaReemplazo()

# Usado en interacion 2 y 3 gráficas, pero cambia de donde se sacan las opciones.
# Retorna una lista con : [Opciones para cambio, sede origen-> Solo aplica para cambio-sede, rol]
@classmethod
def getTandaReemplazo(cls):
    cls.opcionesParaReemplazo = []
    if cls.idxRol < len(cls.rolesAReemplazar):
        sede=None
        cantidad=0
        rol = cls.rolesAReemplazar[cls.idxRol]
        if cls.estadoGestionHumana == "cambio-sede":
            sede = cls.transferirDe[cls.idxRol]
            for emp in sede.getListaEmpleados():
                if emp.getRol() == rol:
                    cls.opcionesParaReemplazo.append(emp)
            cantidad = sum(1 for emp in Main.despedidos if emp.getRol() == rol)
        return cls.opcionesParaReemplazo, sede, rol, cantidad
    else:
        return cls.getTandaContratacion() # Hace lo mismo, pero no toma en cuenta la sede
else:
    return None
```

Interacción 3

Main en su método *prepararContratacion* ordena a la capa lógica que le brinde una lista de personas contratables, es decir, aquellas que no trabajen y cuyo rol lo tenga al menos uno de los empleados a reemplazar, a través del método *Persona.entrevistar*.

Se muestra en pantalla dicha lista para que el usuario pueda elegir a quién o quiénes contratar, asegurándose que se elija la cantidad correcta de cada rol.

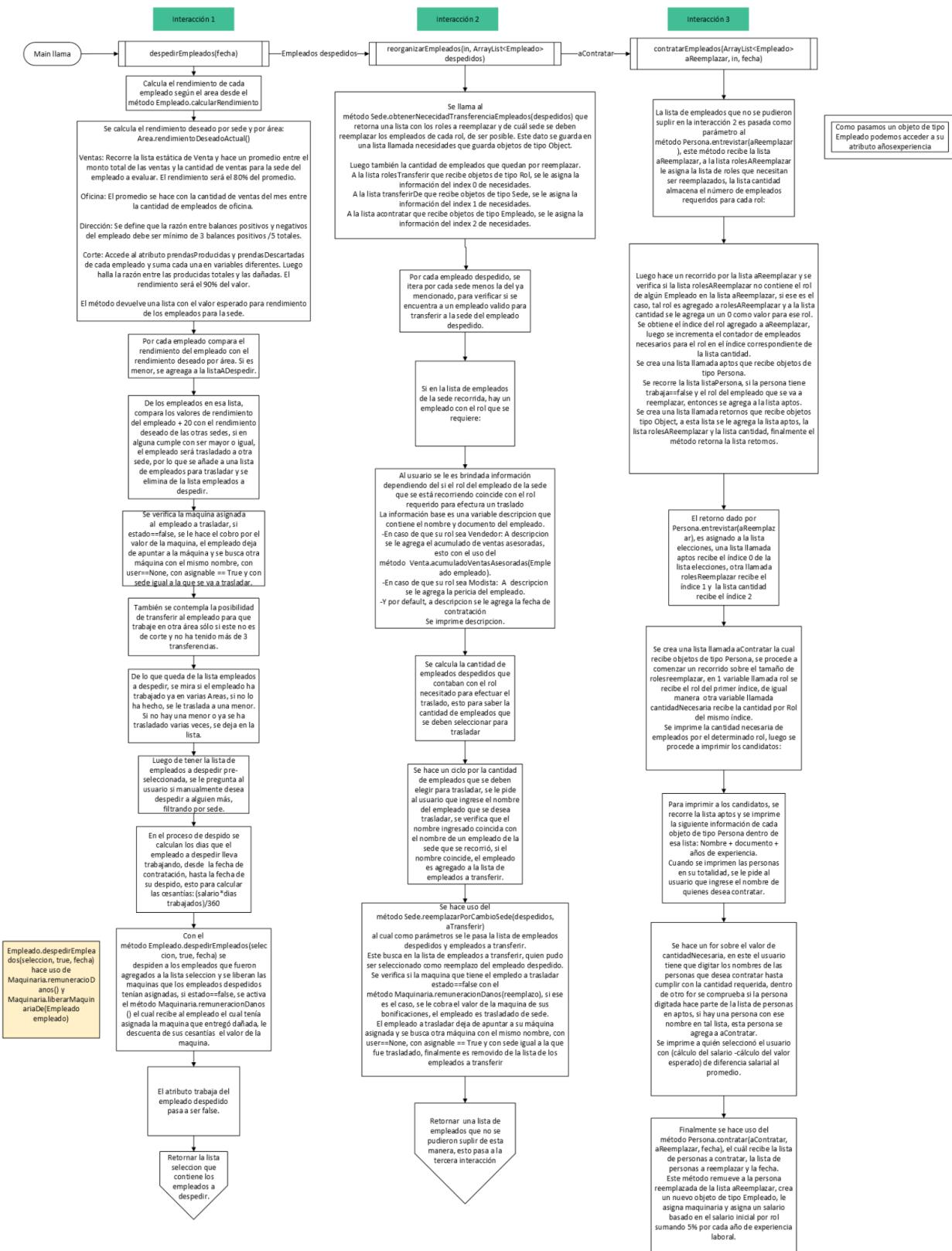
Main ordena a la capa lógica la contratación de la selección del usuario, esto implica añadir a la persona al registro como empleado, asignarle maquinaria y calcular su salario, el salario se calcula con el salario inicial del rol sumando 5% por cada año de experiencia que la persona tenga, esto en el metodo *Persona.contratar*.

```
# Inicia interacción 3 grafica.
@classmethod
def prepararContratacion(cls):
    cls.aptosParaContratar, cls.rolesAReemplazar, cls.cantidadAContratar= Persona.entrevistar(cls.porReemplazar)
    cls.idxRol = 0

# Usado antes de dibujar cada Tanda en interacción 3 grafica.

@classmethod
def getTandaContratacion(cls):
    cls.opcionesParaReemplazo=[]
    for apto in cls.aptosParaContratar:
        if apto.getRol()==cls.rolesAReemplazar[cls.idxRol]:
            cls.opcionesParaReemplazo.append(apto)
    return cls.opcionesParaReemplazo,None, cls.rolesAReemplazar[cls.idxRol], cls.cantidadAContratar[cls.idxRol]
dregion
```

Diagrama de interacción:



Enlace: [Diagrama Gestión Humana.vsdx](#)

Nombre funcionalidad 2 : Insumos

Interacción 1

Inicialmente se hace llamado al método *planificarProduccion*, en la ejecución de este método se hace llamado al método *predecirVentas* de Venta, el cual lleva a cabo una regresión lineal con el objetivo de estimar las ventas del próximo mes, tomando como base los datos de prendas vendidas en cada una de las sedes. Esta aproximación permite identificar tendencias y patrones en el comportamiento de ventas, proporcionando la cantidad de prendas que se venderán el siguiente mes. Una vez generada la proyección, por cada sede, se incorpora un componente de pesimismo para considerar posibles fluctuaciones negativas en las ventas. Este porcentaje de ajuste puede ser *revisado y modificado por el usuario* según su criterio, lo que permite adaptar la proyección a diferentes escenarios de incertidumbre. Tras aceptar o modificar el porcentaje de pesimismo, la predicción de ventas del siguiente mes es calculada a partir de la proyección y el porcentaje de pesimismo y es mostrada en pantalla.

Además, por cada sede, se procesan los datos de la predicción por cada prenda para revisar la cantidad de cada insumo a pedir, con lo cual se crean dos listas específicas: una que contiene los insumos necesarios para la producción y otra que detalla las cantidades requeridas de cada uno. Finalmente, se retornan ambas listas.

```
# Interacción 1
@classmethod
def planificarProduccion(cls,pesimismos): # pesimismos van de 0 a 100,y cambia la predicción en ese porcentaje
    from src.gestorAplicacion.bodega.pantalon import Pantalon
    from src.gestorAplicacion.bodega.camisa import Camisa
    fecha=Main.fecha
    criterios = []
    valores = []
    retorno = []
    Main.texto = []
    cls.pesimismoPorSede = []
    for i in range(len(Sede.getListaSedes())):
        cls.pesimismoPorSede.append(float(pesimismos[i])/100)

    for idxSede,sede in enumerate(Sede.getListaSedes()):
        pantalonesPredichos = False
        camisasPredichas = False
        insumoXSede = []
        cantidadAPedir = []
        listaSede = [insumoXSede, cantidadAPedir]

        for prenda in Sede.getPrendasInventadas(sede):

            if isinstance(prenda, Pantalon) and not pantalonesPredichos:
                proyeccion = Venta.predecirVentas(fecha, sede, prenda.getNombre())
                prediccionP = proyeccion * (1 - cls.pesimismoPorSede[idxSede])
                Main.texto.append(f"La predicción de ventas para {prenda} es de {math.ceil(prediccionP)} para la sede {sede}")
                insumoXSede.append("Pantalon")
                cantidadAPedir.append(prediccionP)
                pantalonesPredichos = True
            elif isinstance(prenda, Camisa) and not camisasPredichas:
                proyeccion = Venta.predecirVentas(fecha, sede, prenda.getNombre())
                prediccionP = proyeccion * (1 - cls.pesimismoPorSede[idxSede])
                Main.texto.append(f"La predicción de ventas para {prenda} es de {math.ceil(prediccionP)} para la sede {sede}")
                insumoXSede.append("Camisa")
                cantidadAPedir.append(prediccionP)
                camisasPredichas = True
```

Interacción 2

A continuación, a partir del método *coordinarBodega* se recorre la lista de insumos por cada sede, y para cada uno de ellos se compara la cantidad requerida del insumo con la cantidad disponible en la *listaInsumosBodega* correspondiente a la sede. Si hay disponibilidad suficiente, se descuenta de la cantidad requerida lo que se toma de la bodega, se ajusta la

cantidad restante del insumo a cero y se ajusta el valor del *precioStockTotal* con el método *restarInsumo* de la clase Sede.

Si con el insumo tomado de la bodega aún no se cumple con la cantidad necesaria para la producción, se verifica en otras sedes si hay el insumo y basándose en la decisión que tome el usuario de si desea transferir o comprar el insumo se sigue el proceso, si aún después transferir el insumo la cantidad no es suficiente y si por otra parte, el usuario decide no transferir, dicho insumo con la respectiva cantidad serán almacenados en la lista a pedir (A).

Finalmente, se retorna la lista de insumos que efectivamente deben pedirse a los proveedores.

```
# Interacción 2
@classmethod
def coordinarBodega(cls): # Antes coordinarBodegas
    insumoFieldFrame = []

    insumoFieldFrame.clear()
    insumosNecesarios = cls.nececidadInsumos[cls.indexSedeCoordinarBodegas][0]
    cantidadesNecesarias = cls.nececidadInsumos[cls.indexSedeCoordinarBodegas][1]

    s=Sede.getListasedes()[cls.indexSedeCoordinarBodegas]

    cls.productosOpcionTransferencia.clear()

    productosAComprar = []
    cantidadesAComprar = []
    for i in insumosNecesarios:
        insumoFieldFrame.append(str(i) + f" ${Insumo.getPrecioIndividual(i)}")
        (hayEnBodega,indiceEnBodega) = Sede.verificarProductoBodega(i, s)
        idxInsumo = insumosNecesarios.index(i)
        filaTabla=[i.getNombre(),0,cantidadesNecesarias[idxInsumo],0]
        cantidadADConseguir=cantidadesNecesarias[idxInsumo]
        if hayEnBodega:
            cantidadEnBodega= Sede.getCantidadInsumosBodega(s)[indiceEnBodega]
            cantidadADConseguir= max(cantidadesNecesarias[idxInsumo] -cantidadEnBodega, 0)
            filaTabla[3]=cantidadADConseguir
            filaTabla[1]=cantidadEnBodega
        if cantidadADConseguir>0:
            productoEnOtraSede = Sede.verificarProductoOtraSede(i,cls.getSedeActualCoordinarBodegas())
```

Interacción 3

Luego, se llama al método *comprarInsumos*, aquí se realiza el proceso que inicia filtrando, por cada insumo, los proveedores que tienen la capacidad de suministrarlo. Posteriormente, se consulta a cada proveedor sobre el precio del insumo y se selecciona la oferta más económica disponible.

Una vez seleccionado el proveedor con el mejor precio, se verifica si este nuevo precio es inferior al precio más reciente al que se adquirió el insumo, valor almacenado en el atributo *últimoPrecio*, en ese caso el sistema pregunta al usuario si desea pedir una cantidad adicional del insumo, ofreciendo la posibilidad de especificar cuánto desea agregar a la cantidad originalmente solicitada, luego se suma dicho valor a la cantidad a pedir.

Después de concretar la compra, gracias al método *añadirInsumo*, se actualiza el inventario de cada sede, incrementando la cantidad disponible del insumo correspondiente según la cantidad adquirida, al mismo tiempo, y por cada insumo, se adquiere una Deuda con el proveedor por el monto de la compra y con la fecha dada, en caso de que la empresa ya cuente con una deuda con dicho proveedor esta deuda ya existente simplemente se actualiza con el monto de la compra.

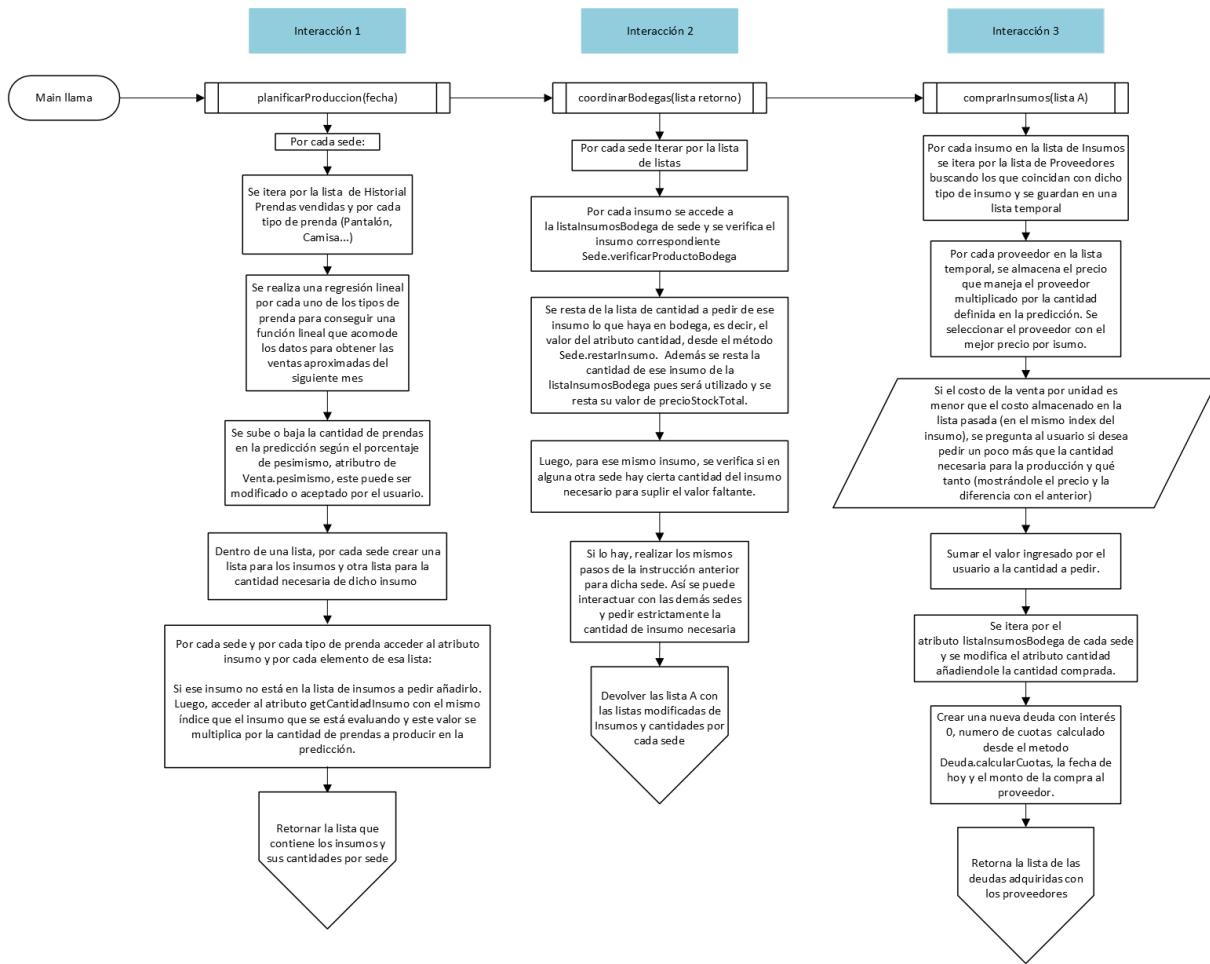
Finalmente, el proceso retorna una lista completa de las deudas generadas durante la operación.

```
# Interacción 3
@classmethod
def comprarInsumos(cls):
    from src.gestorAplicacion.bodega.proveedor import Proveedor
    from src.gestorAplicacion.administracion.deuda import Deuda
    cls.extraPorComprar = []
    criterios=[]
    for idxSede,sede in enumerate(cls.planDeCompra):
        insumos = sede[0]
        cantidad = sede[1]
        insumosPorComprarExtra=[]
        cantidadPorComprarExtra=[]
        quedanPorComprarSede=[insumosPorComprarExtra,cantidadPorComprarExtra]
        for idxInsumo in range(len(insumos)):
            mejorProveedor = None
            mejorPrecio = 0
            insumo:Insumo=insumos[idxInsumo]
            instanciaSede:Sede=Sede.getListaSedes()[idxSede]
            for proveedor in Proveedor.getListaProveedores():
                if Proveedor.getInsumo(proveedor).getNombre() == insumos[idxInsumo].getNombre():
                    if proveedor.getPrecio() < mejorPrecio or mejorPrecio==0:
                        mejorProveedor = proveedor
                        mejorPrecio = proveedor.getPrecio()
            insumo.setProveedor(mejorProveedor)

            insumoEnBodega=instanciaSede.getListaInsumosBodega()[instanciaSede.encontrarInsumoEnBodega(insumo)]

            ultimoPrecio=insumoEnBodega.getUltimoPrecio()
            if mejorPrecio < ultimoPrecio:
                diferencial =ultimoPrecio-mejorPrecio
                criterios.append(f'{insumos[idxInsumo].getNombre()} para {instanciaSede.getNombre()} por ${diferencial}'
```

Diagrama de interacción funcionalidad Insumos:



Enlace: [Diagrama Insumos.vsdx](#)

Nombre funcionalidad 3 : Sistema Financiero

Interacción 1

Inicialmente se le pregunta al usuario qué deudas desea calcular, donde debe escribir si o no según quiera calcular las deudas con Proveedor, Banco o ambos. Además se le pide escoger el directivo que se hará cargo del balance. Se le pasan estos datos al método del main *calcularBalanceAnterior* se hace el llamado al método de Venta *calcularBalanceVentaProduccion* y se le pasa la fecha.

Se calculan los ingresos de las ventas iterando por cada sede en su respectiva lista de ventas y por cada una de ellas se calcula el subtotal del dinero que entró a la empresa este mes, es decir, el monto pagado menos el descuento que se le hizo a la persona por la compra (A). En ese mismo ciclo, se evalúa el costo de producción de cada prenda en base al costo de sus insumos (B). A la suma de todos los ingresos (A) se le resta los costos de la producción de todas la prendas (B) y se devuelve ese valor (C).

Las deudas elegidas por el usuario anteriormente se le pasan al método *calcularDeudaMensual* de Deuda. Luego, el programa ejecuta uno de los casos dependiendo de la elección del usuario. En cada caso se itera sobre la lista de deudas y se llama al método *deudaMensual* para que calcule el valor a pagar de la deuda este mes, teniendo en cuenta el valor que se ha pagado de la deuda original, aplicando intereses y dividiendo entre el número de cuotas faltantes. Para el caso de los proveedores, además de lo anterior, verifica que el proveedor venda alguno de los insumos necesarios para la fabricación de las prendas. Este método devuelve el valor de la suma de todas las deudas mensuales calculadas (D).

Este valor (D) se le resta a (C) y genera el balance total. Este es añadido a la lista de balances de alguno del Empleado de dirección que el usuario eligió, esto afecta su estado para su evaluación en Gestión humana. Finalmente, se retorna dicho balance y se lo muestra al usuario en pantalla. El usuario debe oprimir siguiente para seguir con la segunda interacción.

```
#Interaccion 1
def calcularBalanceAnterior(empleado, eleccion):
    from src.gestorAplicacion.administracion.evalucionFinanciera import EvaluacionFinanciera
    from src.gestorAplicacion.administracion.deuda import Deuda
    from src.gestorAplicacion.administracion.area import Area
    balanceCostosProduccion = Venta.calcularBalanceVentaProduccion(Main.fecha)
    deudaCalculada = Deuda.calcularDeudaMensual(Main.fecha, eleccion)
    balanceTotal = balanceCostosProduccion - deudaCalculada
    nuevoBalance = EvaluacionFinanciera(balanceTotal, empleado)
    Main.nuevoBalance=nuevoBalance
    return nuevoBalance
```

Interacción 2

Se le pide al usuario el porcentaje de fidelidad que desea aplicar a los clientes sin membresía (porcentaje de las ventas que se pueden dar por sentado y se mantienen de un mes al otro) puesto que no son compradores frecuentes. Se recibe entonces el balance previo y el porcentaje de fidelidad para predecir las ventas próximas según su membresía, se recomienda en general 90% para clientes oro, 70% para plata y 50% para bronce, *sin embargo*, este porcentaje es ajustado según el resultado del balance previo para el resto de membresías.

Se hace el llamado al método de `Evaluacionfinanciera estimadoVentasGastos` y se le pasa la fecha, el porcentaje elegido por el usuario y el balance anterior. Allí, se calcula el monto de las ventas del mes pasado para luego aplicarle el porcentaje de fidelidad y predecir así el monto de ventas que se espera para el próximo mes (B). Luego, se evalúan los gastos mensuales promedio llamando al método estático de la interfaz `gastosMensuales` para que calcule los gastos mensuales aproximados de Prenda, Bolsa, Maquinaria y Empleado y devuelva un valor (C).

Después, se halla la diferencia entre la predicción de ventas (A) y la suma de este valor (C) junto con el balance calculado en la anterior interacción para conocer si el 80% de los gastos se pueden cubrir con las ventas estimadas. Por último, se retorna este valor y se muestra en pantalla.

```
# Interaccion 2
def calcularEstimado(porcentaje):
    from src.gestorAplicacion.administracion.evaluacionFinanciera import EvaluacionFinanciera
    diferenciaEstimado = EvaluacionFinanciera.estimadoVentasGastos(Main.fecha, porcentaje, Main.nuevoBalance)
    # Un mes se puede dar por salvado si el 80% de los gastos se pueden ver
    # cubiertos por las ventas predichas
    Main.diferenciaEstimado=diferenciaEstimado
    return diferenciaEstimado
```

Interacción 3

Se le pide al usuario ingresar un banco B para posibles endeudamientos, además, se recibe la fecha y el monto de diferencia calculado entre las ventas estimadas y los gastos mensuales cuyo resultado es el monto necesario para mantener la empresa a flote.

Se tiene una cantidad de deuda a adquirir o pagar (A) que depende de si el estimado es negativo o positivo. Si el balance es negativo, se debe crear una nueva Deuda con dicho monto y fecha de hoy con el Banco elegido. Por el contrario, si el balance es positivo se hará el intento de pagar, a partir de los ahorros, al menos una parte de las dos mayores deudas, una de Proveedores y la otra de Bancos (Se muestran en Pantalla las deudas pagadas).

Llamando al método de venta llamado `blackFriday` se hará el cálculo de un descuento

(B). La posibilidad de tener ofertas dependen de los datos del último 29 de noviembre y fechas aledañas, por lo que lo primero es fijar en una lista las fechas del último black Friday (28,29 y 30 de noviembre) y en otra lista fijar fechas cercanas pero que no se incluyen en la lista anterior (23,24 y 25 de noviembre). Esto se hace con datos de venta. Se filtra la lista de ventas por las fechas del black Friday y otra lista de fechas aleatorias (para hacer la comparación más cercana serán del mismo mes y año). Luego, se compara el monto recogido de las ventas que hubo durante ambos rangos de tiempo para saber cómo reacciona el público ante los descuentos. Si durante las fechas del black Friday hubo un aumento en las ventas, esto significa que la audiencia reacciona bien a los descuentos y según la diferencia entre el monto de las ventas del black Friday y las de los días aleatorios, se asigna un cierto porcentaje (B), que es el que retornará el método.

Este porcentaje de aumento por black friday (B) se le da *al usuario, para que juzgue si considera o no que se amerita este descuento*, junto con una recomendación de si se cree que los descuentos van a funcionar o no. Si lo decide cambiar, este se guarda en la variable *nuevoDescuento* pues se necesita tener los valores de ambos descuentos para el paso siguiente.

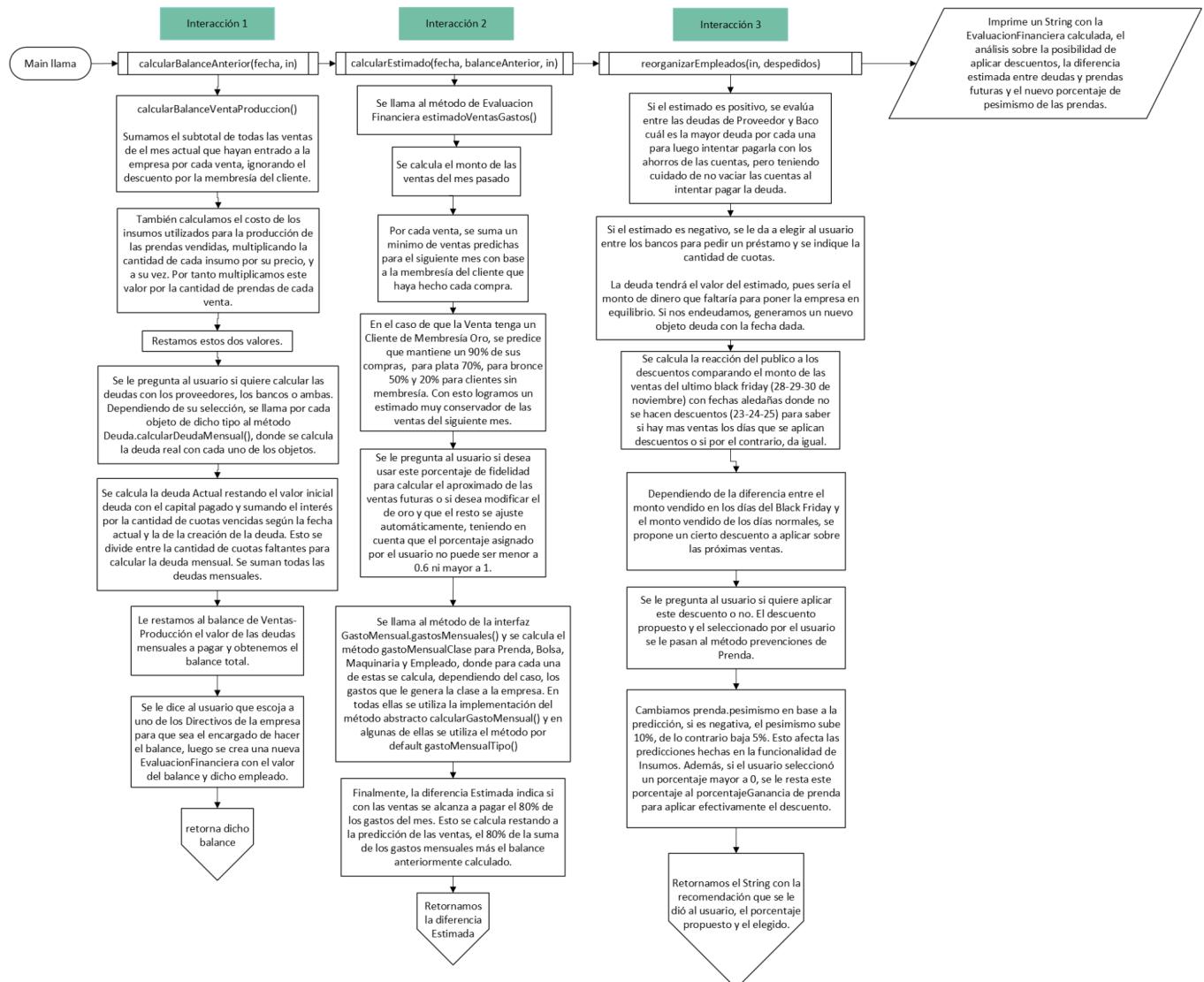
Se accede al método *prevenciones* de Prenda. Allí, se cambia el porcentaje de ganancia de todas las prendas para que así se vea reflejado el descuento elegido por el usuario en esperanzas de subir las ventas. Si el usuario considera que no se ameritan descuentos, entonces no se hacen rebajas, en vez de ello se suma 0.1 a pesimismo, esto debe hacer que al surtir, se hagan 10% menos prendas, si el porcentaje propuesto en el cálculo fue positivo o el usuario dijo que ameritan rebajas, la tasa de pesimismo baja 5% en vez de subir. Se presenta en pantalla la decisión del usuario y la recomendación del programa.

Finalmente, la funcionalidad retorna un String exponiendo el resultado de la evaluación del estado Financiero, mostrando quién hizo el balance y de cuánto fue, el resultado del análisis sobre del black friday, la diferencia estimada entre ingresos/ventas y deudas y el nuevo porcentaje de pesimismo para las prendas.

```
# Interacción 3
def planRecuperacion(diferenciaEstimada, banco):
    from src.gestorAplicacion.bodega import Prenda
    from src.gestorAplicacion.administracion.deuda import Deuda
    bancos=Banco.getListaBancos()
    if diferenciaEstimada > 0:
        deudaPagada= Deuda.compararDeudas(Main.fecha)
        return deudaPagada
    else:
        cuotas = 0
        while cuotas <= 0 or cuotas > 18:
            Deuda.calcularCuotas(diferenciaEstimada)
        deudaAdquirir = Deuda(Main.fecha, diferenciaEstimada, banco, "Banco", cuotas)
        return deudaAdquirir

def descuentosBlackFriday(descuento, nuevoDescuento):
    bfString = None
    if descuento <= 0.0:
        bfString = ("El análisis de ventas realizado sobre el Black Friday arrojó que la audiencia no reacciona bien")
    else:
        bfString = ("El análisis de ventas realizado sobre el Black Friday arrojó que la audiencia reacciona bien")
    Prenda.prevenciones(descuento, nuevoDescuento, Main.fecha)
    analisisFuturo = (f"{bfString}, sin embargo su desición fue aplicar un descuento de: {nuevoDescuento * 100}%.")
    return analisisFuturo
```

Diagrama de interacción:



Enlace: [Diagrama Sistema Financiero.vsdx](#)

Nombre funcionalidad 4 : Facturación

Interacción 1

Inicialmente se le pide al usuario que ingrese la fecha y se le hace llamado al método *ingresarFecha* para registrar el momento en el que se realizará la venta:

Se le pide *al usuario que seleccione al cliente al que se le efectuará la venta* y con el método de la clase *imprimirNoEmpleados*, se muestra al usuario al filtrar cada uno de los objetos de tipo persona que no están cumpliendo con el rol de Empleado y que se pueden elegir como el cliente al que se le realizará el servicio de venta. Se le pide además al usuario que seleccione una de las sedes que aparezcan por pantalla y según eso se le muestran los posibles empleados para Vendedor y Empleado caja de los cuáles debe elegir uno para cada uno (acorde con su rol, uno de los empleados hace parte del área de ventas y se encargará de asesorar la compra del cliente, el otro, el cuál es del área de oficina se encargará de registrar la venta). Por último se le pide la prenda que desea comprar y la cantidad, además de preguntarle si desea añadir más prendas.

Luego, al tener todo lo anterior seleccionado, las prendas elegidas serán sacadas del stock de prendas de la empresa y de la sede respectiva, en el caso de que la sede en la que se encuentra el cliente no tenga la cantidad de unidades que desea, se realizará una búsqueda de las prendas restantes en otras sedes de la empresa, el costo de envío base cuesta 3.000 pesos y por cada prenda más se le van agregando 1.000 pesos al costo de envío, cuando se encuentre la debida cantidad de prendas, estas serán sacadas del stock de su sede originaria y serán transportadas a la sede donde se encuentra el cliente para continuar la venta.

Posteriormente, se hace el debido cálculo del precio total de la venta por la cantidad de productos de un tipo que se hayan elegido, para esto se hace uso del método de clase *PrecioVenta* que se encuentra en las clases Camisa y Pantalon respectivamente, esto para que todas las prendas de tipo Camisa tengan el mismo precio, esto mismo con las prendas de tipo Pantalon, al finalizar el subtotal de la venta por cantidad de productos, se hace el cálculo del IVA(19%) teniendo en cuenta los costos de envío, si el cliente cuenta con un porcentaje de descuento por su membresía, se le hace el debido descuento al monto total a pagar, se crea un objeto de tipo Venta y luego se usa el set de *costosEnvio* para agregarle el costo de envío de la venta, así mismo como su subtotal y el monto que por ahora se designará como *montoPagado*, se le asigna una comisión del 5% del subtotal al empleado del área de ventas y se muestra en pantalla el subtotal. Finalmente, el objeto Venta es retornado.

```

def vender(cliente, sede, encargado, vendedor, productosSeleccionados, cantidadProductos):
    from ..gestorAplicacion.administracion.empleado import Empleado
    from src.gestorAplicacion.bodega.pantalon import Pantalon
    from src.gestorAplicacion.bodega.camisa import Camisa
    from src.gestorAplicacion.administracion.area import Area
    venta = None
    fechaVenta = Main.fecha
    costosEnvio = 0
    for i in range(len(productosSeleccionados)):
        cantidadPrenda= cantidadProductos[i]
        prendaSeleccionada = productosSeleccionados[i]
        cantidadDisponible = 0
        for prenda in Sede.getPrendasInventadasTotal():
            if(Prenda.getNombre(prenda)==Prenda.getNombre(prendaSeleccionada)):
                cantidadDisponible+=1
        Main.manejarFaltantes(sede, cantidadPrenda, cantidadDisponible, prendaSeleccionada.getNombre(), costosEnvio)
    if 0 < cantidadPrenda < len(Sede.getPrendasInventadasTotal()):
        eliminadas = 0
        idxPrenda=0
        while idxPrenda<len(Sede.getPrendasInventadasTotal()):
            if eliminadas >= cantidadPrenda:
                break
            if Sede.getPrendasInventadasTotal()[idxPrenda] == prendaSeleccionada:
                eliminada = Sede.getPrendasInventadasTotal().pop(idxPrenda)
                eliminadas += 1
            i -= 1
            idxPrenda += 1
    sumaPreciosPrendas = 0
    cantidadCamisas = 0
    cantidadPantalon = 0
    for prenda in productosSeleccionados:
        if isinstance(prenda, Camisa):
            sumaPreciosPrendas += Camisa.precioVenta()*cantidadProductos[productosSeleccionados.index(prenda)]
            cantidadCamisas += 1

```

Interacción 2

Se recibe el objeto Venta y se le pide *al usuario que elija el tamaño de bolsa que el cliente quiera*, repitiendo este proceso con un ciclo hasta que la capacidad de las bolsas seleccionadas iguale o supere la cantidad de productos, solo dando la opción de bolsas que estén en stock y se le indica al usuario cuando ya haya superado la cantidad de productos. Después, las bolsas seleccionadas se almacenan en una lista temporal y se quitan de la lista del stock.

Luego de quitar las bolsas pedidas por el usuario de la lista de stock, se evalúa cuál es el tipo de bolsa que tiene menos cantidad en stock. Luego, si esta tiene menos de 10 bolsas en la sede de la compra, se mostrará un mensaje de bolsas insuficientes en pantalla, de lo contrario se notificará que hay suficiente stock. Se le da la opción al usuario de *elegir cuántas bolsas comprar para reponer; si no desea, ingresa cero*. Se retiran los ahorros correspondientes al costo de la bolsa*cantidad y se añaden al stock. Finalmente, el objeto Venta es retornado y se muestra en pantalla la cantidad comprada y el costo.

```

def cantidadActualBolsas(venta, cantidadBolsaGrande,cantidadBolsaMediana,cantidadBolsaPequeña):
    totalPrendas = len(productosSeleccionados)
    insumosBodega = sede.getListainsumosBodega()
    tamañoListaInsumos = sede.getCantidadInsumosBodega()
    bolsasSeleccionadas = []
    capacidadTotal = 0
    bolsasAPedir=cantidadBolsaGrande+cantidadBolsaMediana+cantidadBolsaPequeña
    debeBolsas=0
    for i in range(bolsasAPedir):
        capacidadBolsa = 0
        if cantidadBolsaGrande > 0:
            capacidadBolsa = 8
            cantidadBolsaGrande -= 1
        elif cantidadBolsaMediana > 0:
            cantidadBolsaMediana -= 1
            capacidadBolsa = 3
        elif cantidadBolsaPequeña >0:
            capacidadBolsa = 1
            cantidadBolsaPequeña -= 1

        cantidadDisponible = 0
        capacidadTotal += capacidadBolsa
        tamañoListaInsumos=len(sede.getListainsumosBodega())
        for i in range(tamañoListaInsumos):
            insumo = sede.getListainsumosBodega()[i]
            if isinstance(insumo, Bolsa):
                if insumo.getCapacidadMaxima() == capacidadBolsa:
                    cantidadInsumosBodega=sede.getCantidadInsumosBodega()
                    cantidadDisponible += cantidadInsumosBodega[i]
                    if cantidadDisponible > 0:
                        cantidadInsumosBodega[i] -= 1
                        break

        debeBolsas=max(totalPrendas-capacidadTotal,0)
        venta.getBolsas().append(bolsasSeleccionadas)
        totalVenta = venta.getMontoPagado() + len(bolsasSeleccionadas) * 2000
        venta.setMontoPagado(totalVenta)
    return debeBolsas

```

Interacción 3

Se recibe el objeto venta (Z) y se pregunta *al usuario si se va a usar una tarjeta de regalo y si desea comprar una para futuras ocasiones, en cuyo caso debe ingresar el monto.* Si tiene tarjeta de regalo, ingresa el código en el espacio correspondiente y se verifica que sea una tarjeta válida, es decir, que esté en la lista estática *códigosRegalos*, luego se revisa si el monto a comprar es igual o supera al monto de la tarjeta *montosRegalo*.

Si lo es, el proceso sigue y solo se cobra el excedente, si hay uno. Si el monto a comprar es menor al monto de la tarjeta, se pide al usuario que seleccione uno o más artículos extra y se inicia el proceso de facturación desde 0 con los nuevos artículos o bien que utilice otro método de pago, pues la compra debe cubrir el monto total de su tarjeta.

Por otro lado, si el usuario decidió comprar una nueva tarjeta de regalo, se genera un código aleatorio el cual se muestra en pantalla y los datos son registrados en *codigosRegalo* y *montosRegalo* respectivamente.

Se recibe el objeto Venta y se hace el cálculo del ingreso, los precios sin IVA pero con descuento por membresía del cliente y por tarjetas de regalo redimidas, se suman las compras de tarjetas de regalo nuevas (F) y se añade a la cuenta bancaria de la sede y al monto total de la venta sin IVA.

Se muestra al usuario el nuevo monto en ahorros de la sede y se le pregunta si quiere transferir los fondos a la cuenta principal y qué tantos fondos (porcentaje 20% y 60%), dicha operación cuesta 50.000 pesos. Si decide hacerlo, entonces se transfieren a la cuenta principal los fondos de esta sede menos 50.000. Esta operación es útil porque con la cuenta principal se realizan diferentes tipos de pagos, pero dicha cuenta está específicamente diseñada para acumular pagos reduciendo costos en transferencia. Finalmente, se le muestra al usuario una factura con información relevante de la venta.

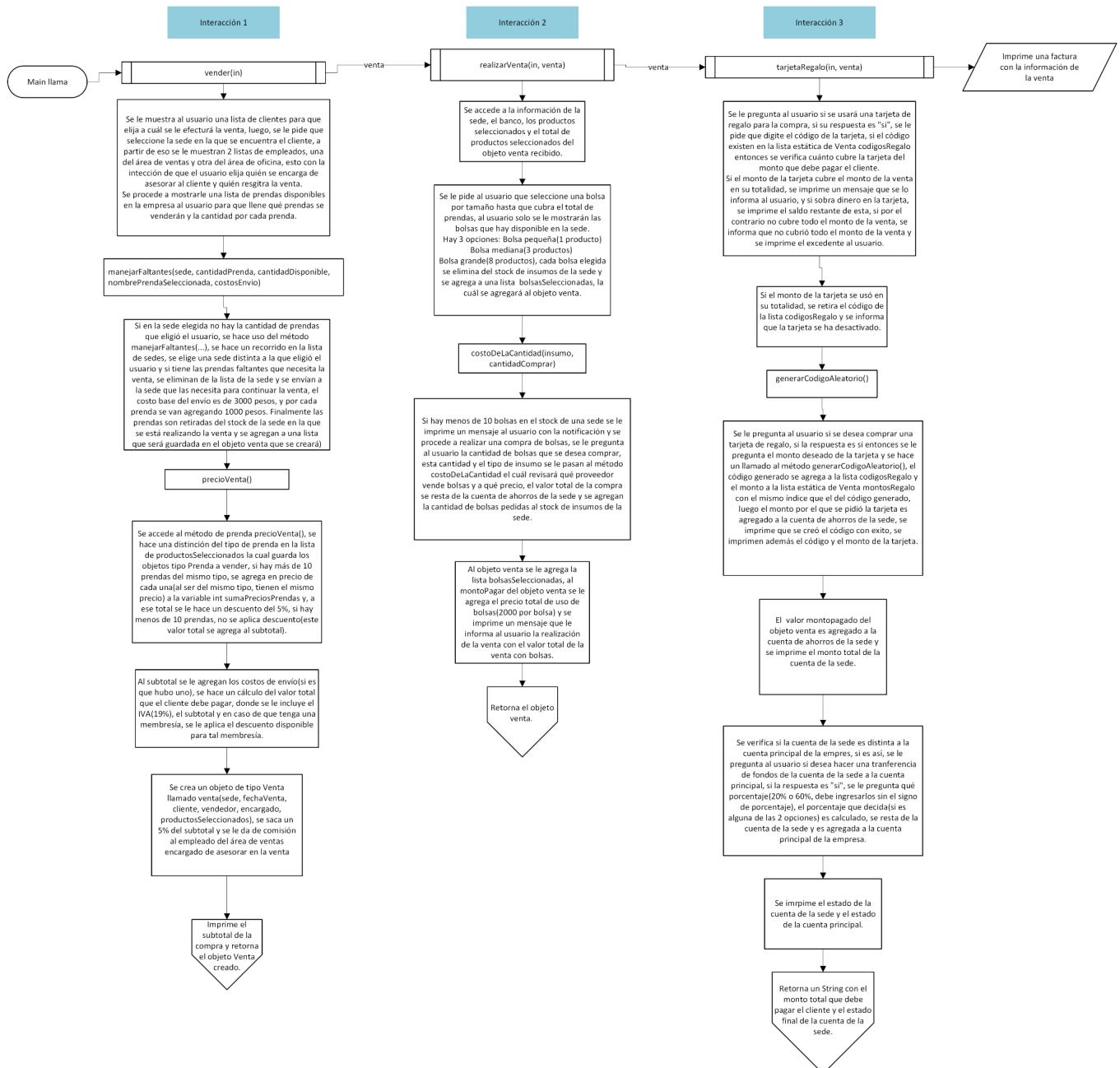
```
def tarjetaRegalo(venta,codigoIngresado,respuesta,compraTarjeta,montoTarjeta):
    nuevoIntento = 1
    while nuevoIntento == 1:
        if respuesta.lower() == "sí":
            if codigoIngresado in Venta.getCodigosRegalo():
                retorno+="Código válido. Procesando tarjeta de regalo..."
                indice = Venta.getCodigosRegalo().index(codigoIngresado)
                montoTarjeta = Venta.getMontosRegalo()[indice]
                montoVenta = Venta.getMontoPagado(venta)

                if montoTarjeta >= montoVenta:
                    retorno+='\nEl monto de la tarjeta cubre la totalidad de la venta.'
                    saldoRestante = montoTarjeta - montoVenta
                    Venta.getMontosRegalo()[indice] = saldoRestante
                    Venta.setMontoPagado(venta,0)
                    retorno+='\nSaldo restante en la tarjeta de regalo: $' + str(saldoRestante)
                else:
                    montoFaltante = montoVenta - montoTarjeta
                    Venta.getMontosRegalo()[indice] = 0
                    venta.setMontoPagado(montoFaltante)
                    retorno+='\nEl monto de la tarjeta no es suficiente para cubrir la venta.'
                    retorno+='\nMonto restante a pagar: $' + str(montoFaltante)

                if Venta.getMontosRegalo()[indice] == 0:
                    Venta.getCodigosRegalo().pop(indice)
                    Venta.getMontosRegalo().pop(indice)
                    retorno+="\nLa tarjeta de regalo se ha agotado y ha sido desactivada."
            nuevoIntento = 2
        else:
            retorno+="\nEl código ingresado no es válido. Por favor, intentar de nuevo o pagar el monto total"
            #print("Ingresa 1 para intentar de nuevo.")
            #print("Ingresa 2 para salir del intento")
            #nuevoIntento = Main.nextIntSeguro()

    elif respuesta.lower() == "no":
        nuevoIntento -= 1
    break
```

Diagrama de interacción:



Enlace: [Diagrama Facturación.vsdx](#)

Nombre funcionalidad 5 : Producción

Interacción 1

Desde el método main se llama al método *agruparMaquinasDisponibles* de la clase Maquinaria. Este recibe como parámetro una fecha específica que representará el contexto temporal de la interacción. Seguidamente se inicializan dos listas:

- maqDisponibles: Para almacenar las máquinas que estarán disponibles tras la validación.
- todosProvBaratos: Para registrar a los proveedores más económicos de repuestos necesarios.

También se declara una variable booleana “encontrado”, que indicará si se logró o no realizar la compra de repuestos.

El método comienza recorriendo la lista de sedes disponibles (*Sede.getlistaSedes()*). Dentro de cada sede, se evalúa la lista de máquinas registradas. Por cada máquina, se determina la diferencia entre las horas predefinidas de revisión de la máquina menos las horas de uso para determinar si se excede las horas programadas para revisión; si las horas restantes son mayores que cero, la máquina no requiere mantenimiento preventivo y el atributo mantenimiento se marca como false.

Seguidamente, por cada repuesto de la máquina, se calcula su estado evaluando las horas de vida útil menos las horas de uso de dicho repuesto; si el cálculo resulta menor o igual a cero, significa que el repuesto ha agotado su vida útil. Se imprime un mensaje indicando que el repuesto necesita ser reemplazado.

Para encontrar el repuesto más barato, se llama al método *encontrarProveedoresBaratos()*, que recorre la lista de proveedores (*Proveedor.getListaProveedores()*). Se realiza una comparación entre los precios individuales de los insumos registrados por cada proveedor. El algoritmo selecciona el proveedor con el precio más bajo utilizando las siguientes comparaciones: Si no hay un proveedor registrado aún, el actual se guarda como el más barato; pero si ya hay un proveedor registrado, se compara su precio con el del actual y se actualiza si el nuevo es más económico.

Una vez identificado el proveedor más barato, el sistema verifica si alguna sede tiene suficiente dinero en su cuenta (*Sede.getCuentaSede().getAhorroBanco()*). Si alguna de las dos sedes cuenta con los fondos necesarios, se llama al método *dondeRetirar()* de la clase Main, el cual solicita al usuario seleccionar de cuál sede descontar el monto del repuesto y, con esto, se procede a comprar y actualizar el saldo de la sede seleccionada.

El repuesto defectuoso se elimina de la lista de repuestos de la máquina y se reemplaza por una copia del nuevo repuesto adquirido. Este nuevo repuesto registra el precio de compra y la fecha en la que fue adquirido.

Tras la evaluación de los repuestos, se verifica si todos están en buen estado. Si todos los repuestos son funcionales y la máquina no requiere mantenimiento, su estado se marca como “true”, lo que hace que se agregue a la lista maqDisponibles.

Si ninguna sede tiene fondos suficientes para adquirir los repuestos, se imprime un mensaje indicando que la máquina queda inhabilitada, ya que los repuestos defectuosos no pudieron reemplazarse.

Una vez finalizado el proceso para todas las sedes y máquinas, el método retorna la lista maqDisponibles. Esta contiene todas las máquinas que están listas para su uso tras las evaluaciones.

```
@classmethod
def agruparMaquinasDisponibles(cls):
    from src.gestorAplicacion.bodega.repuesto import Repuesto
    from src.uiMain.main import Main
    from src.uiMain.startFrame import StartFrame
    from src.gestorAplicacion.bodega.proveedor import Proveedor
    from src.gestorAplicacion.bodega.insumo import Insumo

    #print(f"los repuestos mas actuales creados: {Repuesto.getListadoRepuestos()}\n y hay en total: {len(Repuesto.getListadoRepuestos())}")
    maqDisponibles = []
    todosProvBaratos = []
    encontrado = False
    proveedorBarato = None
    maquinasPaRevisar = []
    for cadaSede in Sede.getListasSedes():
        for cadaMaquina in cadaSede.getListasMaquinas():
            if (cadaMaquina.getHoraRevision() - cadaMaquina.getHorasUso()) > 0:
                for cadaRepuesto in cadaMaquina.getRepuestos():
                    if (cadaRepuesto.getHorasDeVidaUtil() - cadaRepuesto.getHorasDeUso()) <= 0:
                        cadaMaquina.mantenimiento=True
                        todosProvBaratos = Main.encontrarProveedoresBaratos()
                        for elMasEconomico in todosProvBaratos:
                            if elMasEconomico.getInsumo().getNombre().lower() == cadaRepuesto.getNombre().lower():
                                proveedorBarato = elMasEconomico
                                break
                            StartFrame.recibeProveedorB(proveedorBarato)
                            cls.infoRepuestosAComprar.append([cadaRepuesto, proveedorBarato,cadaMaquina])
                        cadaMaquina.ultFechaRevision = cls.fecha

                    repuestosBuenos = 0
                    for rep in cadaMaquina.getRepuestos():
                        if rep.isEstado():
                            repuestosBuenos += 1
                    if len(cadaMaquina.getRepuestos()) == repuestosBuenos:
                        cadaMaquina.estado = True
                    else:
                        cadaMaquina.estado = False
                    maqDisponibles.append(cadaMaquina)
```

Interacción 2

Desde el método main se invoca al método *planProducción*, de la clase Sede, que recibe como parámetros la lista de máquinas disponibles (maqDisponible) retornada de la anterior interacción, la fecha actual (recibida cuando se ejecuta el main) y un objeto Scanner para capturar decisiones del usuario.

Se crean listas para almacenar la producción final (aProducirFinal), lo que se producirá (aProducir) y lo que se pospondrá para producir el día siguiente, por casos de fuerza mayor (listaEspera).

Las máquinas disponibles se dividen por sede en maqSedeP (Sede Principal) y maqSede2 (Sede 2). Como también, las máquinas de cada sede se separan en maqProducción (máquinas de producción) y maqOficina (máquinas no operativas). Despues de esto, se verifica si cada sede tiene al menos tres máquinas de producción: Si una sede cumple este requisito, se asigna una señal: senal = 5 para la Sede Principal y senal += 10 para la Sede 2. Estas señales definen las acciones a seguir:

- senal = 5: Solo Sede Principal puede producir.
- senal = 10: Solo Sede 2 puede producir.
- senal = 15: Ambas sedes pueden producir.
- senal = 0: Ambas sedes están inhabilitadas.

Seguidamente, se llama al método calcProduccionSedes(), que utiliza el método predecirVentas() de la clase Venta para calcular cuántos pantalones y camisas producir en cada sede, usando una regresión lineal basada en las ventas de los tres meses anteriores, de la siguiente manera:

- Ventas futuras = ventas del último mes + pendiente promedio
- La pendiente promedio se calcula así:
$$((\text{Ventas del mes 2} - \text{Ventas del mes 1}) + (\text{Ventas del mes 3} - \text{Ventas del mes 2})) / 2$$

Dependiendo de la señal, el sistema solicita al usuario decidir si:

- Producir todo en una sede.
- Distribuir la producción entre las dos sedes.
- Posponer parte de la producción a un día posterior.

Se actualizan las listas de producción (aProducir) y de espera (listaEspera) según las decisiones tomadas por el usuario, de las anteriormente mencionadas.

Cabe destacar que, si una sede está sobrecargada (más de 10 prendas por modista), se ofrecen dos opciones:

- Redistributions la producción con la otra sede, dividiendo proporcionalmente, es decir, el total de prendas a producir / 2, por lo que son solo dos sedes con las que cuenta la empresa.
- O asumir el costo de sobrecarga y producir todo el mismo día en la sede sobrecargada.

Se registra en aProducirFinal la producción por sede y las prendas en lista de espera. Si ninguna sede está sobrecargada, se realiza la producción normalmente, es decir, sin poner ninguna prenda en lista de espera, ni enviarla a otra sede.

Este metodo retorna aProducirFinal, el cual es un ArrayList que guarda ArrayList de

ArrayList de enteros, en donde la posición 0 de estas listas de enteros siempre será para pantalones y la 1 para camisas. `aProducirFinal` sería la lista que contiene toda la planificación, incluyendo la producción distribuida y lo que se pospondrá, además, todo muy bien separado por tipo de prenda y sede, para que en la interacción 3 se creen las prendas ya como objetos listos para la venta.

```

@classmethod
def planProduccion(cls, maqDisponible: List, fecha: 'Fecha', containerBig, cont, field_frame, labelTotalGastado, cont2,
    from .bodega.maquinaria import Maquinaria
    from src.uiMain.main import Main
    import math
    from src.uiMain.startFrame import StartFrame
    aProducirFinal = []; aProducir = []; listaEspera = []; listaDeCeros = [0, 0]
    listaEsperaVacia = [listaDeCeros.copy(), listaDeCeros.copy()]
    maqSedeP = []; maqSede2 = []
    senal = 0

    # Dividir las máquinas disponibles por sedes
    for todMaquinas in maqDisponible:
        if todMaquinas.getSede().getNombre().lower() == "sede principal":
            maqSedeP.append(todMaquinas)
        else:
            maqSede2.append(todMaquinas)

    # Dividir las máquinas de cada sede por función
    Sede.getListasSedes()[0].maqProduccion = [] ; Sede.getListasSedes()[1].maqProduccion = []
    for todMaqSedeP in maqSedeP:
        if todMaqSedeP.esDeProducción():
            if not todMaqSedeP.mantenimiento:
                Sede.getListasSedes()[0].maqProduccion.append(todMaqSedeP)
            else:
                Sede.getListasSedes()[0].maqOficina.append(todMaqSedeP)
    for todMaqSede2 in maqSede2:
        if todMaqSede2.esDeProducción():
            if not todMaqSede2.mantenimiento:
                Sede.getListasSedes()[1].maqProduccion.append(todMaqSede2)
            else:
                Sede.getListasSedes()[1].maqOficina.append(todMaqSede2)
    if len(Sede.getListasSedes()[0].maqProduccion) > 3:
        senal = 5
    if len(Sede.getListasSedes()[1].maqProduccion) > 3:
        senal += 10

```

Interacción 3

Desde el método `main` se llama a `producirPrendas` de la clase `Prenda`, pasando como parámetros el plan de producción retornado de la interacción anterior y la fecha actual. Este método organiza todo el proceso de producción y devuelve un valor booleano que indica si los insumos fueron suficientes para completar el plan. Según el resultado, se imprime un mensaje que detalla cuántas prendas fueron producidas con éxito y si hubo limitaciones debido a la falta de insumos.

Dentro de `producirPrendas`, se inicializan las variables `cantidadUltimaProducción` para contar las prendas terminadas y `alcanzaInsumos` como indicador de recursos suficientes. Se recorre día a día el plan de producción, que es una lista tridimensional, dividiendo las tareas por sede. Para cada sede en un día específico, se invoca el método `producirListaPrendas`, encargado de gestionar la creación de las prendas y el uso de los insumos.

En `producirListaPrendas`, se calculan las cantidades de pantalones y camisas a producir en base a las instrucciones del plan. Se obtienen los insumos necesarios mediante

insumosPorNombre y se verifica si estos alcanzan para cumplir la demanda. Por cada prenda que se intenta producir, se calcula el total de insumos requeridos multiplicando la cantidad especificada en getCantidadInsumo() por el precio unitario de cada insumo. Si los insumos disponibles no son suficientes, se detiene la producción de esa prenda y se marca alcanzaInsumos como falso.

Una vez creadas las prendas, se clasifican según las etapas de producción requeridas. El método siguientePaso determina qué maquinaria y tiempo se necesitan para avanzar en cada prenda. Estas etapas incluyen corte, tijereado, costura, bordado, termofijado y planchado, con tiempos asignados según el tipo de prenda y la etapa actual. Por ejemplo, un pantalón requiere 5 minutos en corte y 10 minutos en costura, mientras que una camisa puede necesitar 5 minutos adicionales en el proceso de bordado.

Para procesar las prendas, el sistema solicita al usuario seleccionar un modista mediante pedirModista. Cada modista tiene un nivel de pericia que afecta la probabilidad de éxito en las tareas asignadas. La probabilidad de éxito se calcula así: Pericia del modista * Factor de la etapa; donde el factor disminuye en etapas críticas como costura y termofijado. Por ejemplo, un modista con pericia 0.95 tendrá una probabilidad de 0.855 en la etapa de costura si el factor es 0.9.

Seguidamente, las prendas se procesan en tandas utilizando la maquinaria correspondiente. Cada máquina registra el tiempo de uso, calculandolo así: tiempo requerido por prenda * número de prendas en la tanda; y con esto los modistas actualizan su registro de prendas producidas o descartadas. Si una prenda es descartada, esto ocurre debido a que un valor aleatorio supera la probabilidad de éxito calculada.

Una vez finalizado el procesamiento diario, se calcula el costo de producción sumando los salarios de los modistas activos y aplicando un porcentaje del 1%:

Por otra parte, el costo total de una prenda incluye este costo más el costo de insumos, y el costo de insumos se calcula multiplicando la cantidad del insumo por su precio unitario.

Finalmente, el precio de venta de cada prenda se calcula sumando un margen de ganancia al costo total, es decir, (costo total * (1 + Porcentaje de ganancia)).

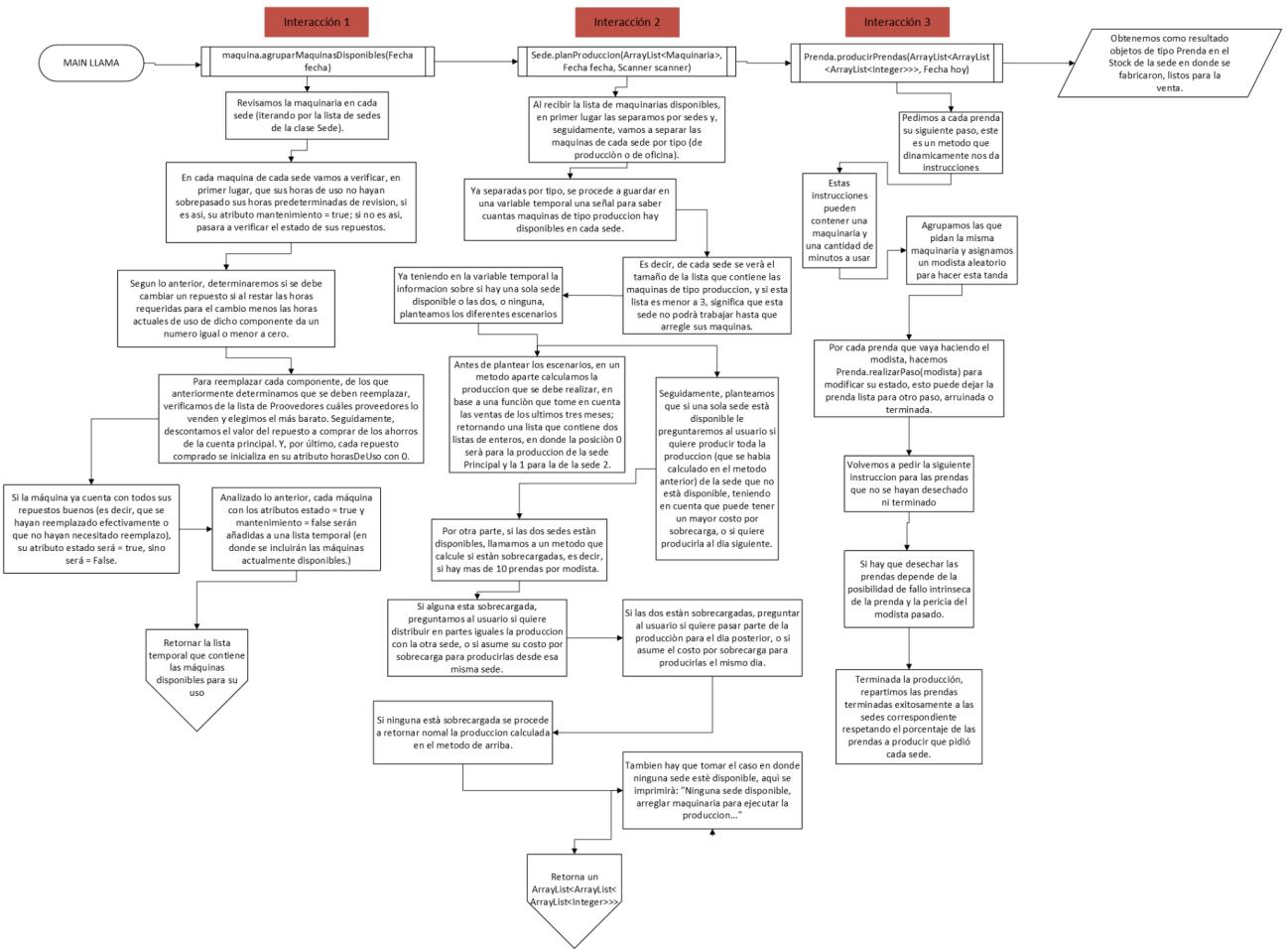
Al concluir, el sistema reporta la cantidad de prendas exitosamente producidas y si hubo limitaciones por falta de insumos. Estos cálculos aseguran que la producción sea eficiente, que los costos se mantengan controlados y que las prendas tengan un precio competitivo en el mercado.

```
@classmethod
def comprarInsumos(cls):
    from src.gestorAplicacion.bodega.proveedor import Proveedor
    from src.gestorAplicacion.administracion.deuda import Deuda
    cls.extraPorComprar = []
    criterios=[]
    for idxSede,sede in enumerate(cls.planDeCompra):
        insumos = sede[0]
        cantidad = sede[1]
        insumosPorComprarExtra=[]
        cantidadPorComprarExtra=[]
        quedanPorComprarSede=[insumosPorComprarExtra,cantidadPorComprarExtra]
        for idxInsumo in range(len(insumos)):
            mejorProveedor = None
            mejorPrecio = 0
            insumo:Insumo=insumos[idxInsumo]
            instanciaSede:Sede=Sede.getListaSedes()[idxSede]
            for proveedor in Proveedor.getListaProveedores():
                if Proveedor.getInsumo(proveedor).getNombre() == insumos[idxInsumo].getNombre():
                    if proveedor.getPrecio() < mejorPrecio or mejorPrecio==0:
                        mejorProveedor = proveedor
                        mejorPrecio = proveedor.getPrecio()
            insumo.setProveedor(mejorProveedor)

            insumoEnBodega=instanciaSede.getListaInsumosBodega()[instanciaSede.encontrarInsumoEnBodega(insumo)]

            ultimoPrecio=insumoEnBodega.getUltimoPrecio()
            if mejorPrecio < ultimoPrecio:
                diferencial =ultimoPrecio-mejorPrecio
                criterios.append(f'{insumos[idxInsumo].getNombre()} para {instanciaSede.getNombre()} por ${diferencial} meno')
                insumosPorComprarExtra.append(insumos[idxInsumo])
                cantidadPorComprarExtra.append(cantidad[idxInsumo])
            else:
                cls.comprarInsumo(cantidad[idxInsumo], insumos[idxInsumo], mejorProveedor, Sede.getListaSedes()[idxSede])
            cls.extraPorComprar.append(quedanPorComprarSede)
    return criterios
```

Diagrama de interacción:



Enlace: [Diagrama Producción.vsdx](#)

5. Manejo de excepciones

La clase llamada ErrorAplicación hereda de la clase Exception para poder desarrollar excepciones personalizadas, las clases ExceptionC1 y ExceptionC2 heredan de ErrorAplicación y en sus módulos se encuentran las respectivas excepciones que heredan de ellas y que se explican a continuación

Excepción de Valor No Válido

```
class ExpcionValorNoValido(ExceptionC1):
    def __init__(self, valor):
        self.mensajeValor=f" La(s) entrada(s) {valor} no contiene(n) un valor válido, por favor, intentar de nuevo "
        super().__init__(self.mensajeValor)
```

Esta excepción hereda de la clase ExceptionC1 y es lanzada cuando un valor no cumple con los requerimientos para ser válido, por ejemplo, un mes debe estar en el rango $0 < \text{mes} \leq 12$, si se escribe un número que no cumpla con ese rango se lanza la excepción y se borra el valor ingresado incorrectamente, este proceso finalizará cuando el usuario llene un valor válido. Se dispara en la interacción 3 de sistema financiero en donde verifica si la entidad bancaria ingresada en el fieldFrame existe.

```
try:
    bancos_disponibles = [Banco.getNombreEntidad(banco) for banco in Banco.getListabancos()]
    if seleccion not in bancos_disponibles:
        raise ExpcionValorNoValido(seleccion)
except ExpcionValorNoValido as ch:
    messagebox.showwarning(title="Alerta", message = ch.mensaje_completo)
    return True
```

Excepción de Contenido Vacío

```
class ExpcionContenidoVacio(ExceptionC1):
    def __init__(self, entradas):
        self.mensajeEntradas=f" Necesita llenar la(s) entrada(s) {entradas} para continuar"
        super().__init__(self.mensajeEntradas)
```

Esta excepción hereda de la clase ExceptionC1 y es lanzada cuando un campo de entrada no es llenado por el usuario, lo cual es de carácter obligatorio en todas las instancias que se presenta, para garantizar que el programa se desarrolle con normalidad, por ello, hasta que todos los campos no sean llenados, el usuario no podrá continuar. Se implementa en la ventana inicial en el espacio habilitado para ingresar la fecha para verificar que sean llenados todos los campos requeridos.

```

if not FDia or FDia == -1 or FDia == " ":
    camposVacios.append("Día")
if not FMes or FMes == -1 or FMes == " ":
    camposVacios.append("Mes")
if not FAño or FAño == -1 or FAño == " ":
    camposVacios.append("Año")
try:
    hayExcepcion = False
    if camposVacios:

        hayExcepcion = True
    if hayExcepcion:
        raise ExcepcionContenidoVacio(camposVacios)
except ExcepcionContenidoVacio as viejaMetida:
    messagebox.showwarning(title="Alerta", message=viejaMetida.mensaje_completo)
    self.borrar()
    return hayExcepcion

```

Excepción de String No Número

```

class ExcepcionStringNoNumero(ExceptionC1):
    def __init__(self, entero):
        self.mensajeEntero=f" La entrada {entero} no es válida, debe llenar este campo solo con String"
        super().__init__(self.mensajeEntero)

```

Esta excepción hereda de la clase ExceptionC1 y es lanzada cuando en una entrada se requiere un dato de tipo String pero es completada con algo distinto, se borra el valor que fue ingresado de manera incorrecta y el proceso finaliza cuando el usuario llena con un valor válido la entrada. Se dispara en la interacción 3 de sistema financiero en donde verifica que el usuario ingrese una cadena de texto.

```

try:
    if seleccion == "":
        raise ExcepcionContenidoVacio(["Bancos"])
    if not isinstance(seleccion, str):
        raise ExcepcionStringNoNumero(seleccion)
except ExcepcionStringNoNumero as p:
    messagebox.showwarning(title="Alerta", message=p.mensaje_completo)
    return True
except ExcepcionContenidoVacio as cabezaHueca:
    messagebox.showwarning(title="Alerta", message=cabezaHueca.mensaje_completo)
    return True

```

Excepción de Número No String

```
class ExpcionNumeroNoString(ExceptionC1):
    def __init__(self, string):
        self.mensajeEntero=f" La entrada {string} no es válida, debe llenar este campo solo con números"
        super().__init__(self.mensajeEntero)
```

Esta excepción hereda de la clase ExceptionC1 y es lanzada cuando se requiere llenar el campo con números enteros pero en su lugar se llena con algo distinto a este requerimiento, se borrará el valor incorrecto ingresado y el proceso dará fin cuando el usuario llene un valor válido. Se dispara en la interacción 2 de sistema financiero en donde verifica que el usuario ingrese un número.

```
try:
    if Porcentaje == "":
        raise ExpcionContenidoVacio(["Fidelidad"])
    elif isinstance(Porcentaje, str) and not str(Porcentaje).replace(".", "", 1).isdigit():
        raise ExpcionNumeroNoString(Porcentaje)
except ExpcionNumeroNoString as uwu:
    messagebox.showwarning(title="Alerta", message=uwu.mensaje_completo)
    return True
except ExpcionContenidoVacio as cabezaHueca:
    messagebox.showwarning(title="Alerta", message=cabezaHueca.mensaje_completo)
    return True
```

Excepción Código de Tarjeta de regalo

```
class ExpcionCodigoTarjetaregalo(ExceptionC2):
    def __init__(self, codigo):
        self.mensajeCodigo = f" El código {codigo} no se encuentra habilitado. ¿Desea intentar nuevamente?"
        super().__init__(self.mensajeCodigo)
```

Esta excepción hereda de la clase ExceptionC2 y es lanzada cuando el usuario llena el campo de validación de tarjeta de regalo con un código que se encuentra deshabilitado o bien nunca existió, se le preguntará si desea intentar llenar el campo del código de la tarjeta de regalo nuevamente, si presiona el botón "Si" se le permitirá volver a llenar el campo, si presiona el botón "No", el programa continuará su flujo de funcionamiento. Se dispara en la tercera interacción de facturación en la cual el usuario debe ingresar el código de una tarjeta de regalo.

```
try:
    raise ExpcionCodigoTarjetaregalo(self.datosEntradasFacturacion.getValue("Código"))
except ExpcionCodigoTarjetaregalo as chacarron:
    messagebox.showwarning(title="Alerta", message=chacarron.mensaje_completo)
    return True
```

Excepción de Prenda No Existente

```
class ExpcionPrendaNoExistente(ExceptionC2):
    def __init__(self, prenda):
        self.mensajePrenda = f" La prenda {prenda} no es vendida en nuestra empresa. Intente nuevamente"
        super().__init__(self.mensajePrenda)
```

Esta excepción hereda de la clase ExceptionC2 y es lanzada cuando la prenda ingresada para realizar una venta es diferente a las prendas camisa o pantalón, las cuales son las únicas que ofrece nuestro sistema. Se dispara en la primera interacción de facturación para verificar que la prenda ingresada en efecto exista en el sistema.

```
try:
    if self.datosEntradasFacturacion.getValue("Prenda").lower() not in ["camisa","pantalon"]:
        raise ExpcionPrendaNoExistente(self.datosEntradasFacturacion.getValue("Prenda"))
except ExpcionPrendaNoExistente as b:
    messagebox.showwarning(title="Alerta", message=b.mensaje_completo)
else:
```

Excepción de Empleado No Encontrado

```
class ExpcionEmpleadoNoEncontrado(ExceptionC2):
    def __init__(self):
        self.mensajeEmpleado = f"Empleado no valido","Verifique que el empleado trabaja en la empresa."
        super().__init__(self.mensajeEmpleado)
```

Esta excepción hereda de la clase ExceptionC2 y es lanzada cuando se llena un campo con el nombre de una persona que no se encuentra en el sistema como empleado de la empresa. Se dispara en la última interacción de gestión humana, verifica que el nombre ingresado en el campo del fieldFrame cumpla con lo mencionado anteriormente.

```
try:
    empleadoExpcion = False
    if existen:
        Main.estadoGestionHumana="cambio-sede"
        self.reemplazarPorCambioSede()
        empleadoExpcion = False
    else:
        empleadoExpcion = True
    if empleadoExpcion:
        raise ExpcionEmpleadoNoEncontrado()
except ExpcionEmpleadoNoEncontrado as lol:
    messagebox.showwarning(title="Alerta", message=lol.mensaje_completo)
return empleadoExpcion
```

Excepción de Agregar Otra Prenda

```
class ExpcionAgregarOtraPrenda(ExceptionC2):
    def __init__(self):
        self.mensajeCompra = f"Desea agregar más prendas a la compra?"
        super().__init__(self.mensajeCompra)
```

Esta excepción hereda de la clase ExceptionC2 y es lanzada para verificar si el usuario tiene la intención de agregar otra prenda a la venta, en caso de presionar el botón "Si" se efectuará la nueva elección de la prenda que será agregada a la venta, si presiona el botón "No", el programa continuará su flujo de funcionamiento. Se dispara en la primera interacción de facturación para que el usuario pueda agregar más de una prenda a la compra a realizar.

```
try:
    if excepcion:
        raise ExpcionAgregarOtraPrenda()
except ExpcionAgregarOtraPrenda as e:
    excepcion = messagebox.askyesno(title = "Confirmación", message= e.mensaje_completo)
```

6. FieldFrame

La clase FieldFrame extiende la clase Frame de Tkinter, proporcionando un contenedor personalizado que permite mostrar criterios y sus respectivos campos de entrada (Entry), con opciones adicionales para botones de aceptación y borrado. Esta clase facilita la creación de formularios dinámicos dentro de la interfaz gráfica.

Implementación y componentes

Constructor `__init__`

frame: Frame padre donde se implementara el FieldFrame.

tituloCriterios: Título para la columna de criterios.

criterios: Lista de etiquetas que describen cada campo.

tituloValores: Título para la columna de valores.

valores: Lista opcional de valores por defecto que se cargarán en los campos de entrada.

habilitado: Lista opcional de booleanos que indica si cada campo debe estar habilitado o deshabilitado.

ancho_entry: Ancho de los campos de entrada.

crecer: Indica si el FieldFrame debe expandirse dinámicamente.

tamañoFuente: Tamaño de la fuente para las etiquetas y botones.

aceptar: Si es True, añade un botón "Aceptar".

borrar: Si es True, añade un botón "Borrar".

callbackAceptar: Función a ejecutar cuando se presiona el botón "Aceptar".

```
class FieldFrame(Frame):
    def __init__(self, frame, tituloCriterios, criterios, tituloValores, valores=None, habilitado=None, ancho_entry=20):
        super().__init__(frame)
        self.valores = []
        self.valoresPorDefecto = valores
        self.criterios = criterios
        self.crecer = crecer
        self.tamañoFuente = tamañoFuente
        self.callbackAceptar = callbackAceptar
        self.createWidgets(tituloCriterios, criterios, tituloValores, valores, habilitado, ancho_entry, aceptar, borrar)
```

Método `createWidgets`:

Genera dinámicamente las etiquetas y campos de entrada, configura la disposición de las columnas, inserta valores por defecto si están definidos y añade botones "Aceptar" y "Borrar" si están habilitados.

Método `aceptar`:

Obtiene los valores ingresados, verifica si hay campos vacíos y lanza una excepción personalizada (ExcepcionContenidoVacio) si hay campos vacíos, mostrando una advertencia.

Método habilitarEntry:

Permite habilitar o deshabilitar un campo específico según el criterio asociado.

Método getValue:

Devuelve el valor actual de un campo según el criterio asociado.

Método obtenerTodosLosValores:

Retorna una lista con los valores actuales de todos los campos.

Método configurarCallBack:

Asigna una función específica a un evento determinado para un campo específico.

Método borrar:

Restaura los valores por defecto si existen y si no hay valores por defecto, vacía todos los campos.

Alcance y funcionalidad:

La clase FieldFrame tiene un alcance directo dentro de la interfaz gráfica, permitiendo la creación flexible de formularios interactivos. Es especialmente útil para gestionar entradas de datos dinámicas, permitiendo a los usuarios modificar valores y ejecutar acciones (como aceptar o borrar) según las necesidades del programa.

El uso de esta clase asegura una organización clara de la interfaz y permite personalizar la experiencia del usuario mediante la configuración dinámica de campos y callbacks.

Implementación

```
def seleccionadorDespedidos(self):
    valores=[self.cantidadADespedir]
    criterios=["Cantidad de despedidos"]
    for i in range(self.cantidadADespedir):
        criterios.append(f"Nombre del despedido {i+1}")
        valores.append("")
    self.seleccionador=FieldFrame(self.frameCambianteGHumana, "Dato",
                                    criterios, "valor",valores, ancho_entry=20,
                                    tamañoFuente=10,aceptar=True, borrar=True, callbackAceptar=self.despedir)
    self.seleccionador.configurarCallBack("Cantidad de despedidos", "<Return>", lambda e:self.actualizarCantidadDespedidos)
    self.seleccionador.grid(row=2, column=0,columnspan=1)
```

En la clase startFrame en el método seleccionarDespedidos que hace parte de la segunda interacción de la funcionalidad de gestión humana se puede evidenciar una instancia de fieldFrame que se utiliza para realizar el proceso de despido de empleados, en el cual al presionar el botón de aceptar se despiden a los seleccionados.

```

# Interacción 1
def pesimismo(self, c, v):
    from src.uiMain import fieldFrame
    criterios = c
    valores = v

    self.frame2 = tk.Frame(self.framePrincipal, bg="light gray")
    self.frame2.pack(anchor="s", fill="x")

    self.field = fieldFrame.FieldFrame(self.frame2, "\nPuede cambiar la predicción de ventas para el siguiente mes",
                                         criterios, "El porcentaje de pesimismo es de", valores, [True, True], 20,
                                         False, 10, True, False, lambda : self.prediccion(Main.texto, Main.retorno))
    self.field.pack(anchor="s", expand=True, fill="both")

```

En el método pesimismo de la clase startFrame se puede encontrar otro uso del FieldFrame para que el usuario pueda cambiar el porcentaje de pesimismo para cada sede y a través del botón aceptar se hace llamado del método predicción.

```

def interaccion1Facturacion(self):
    self.descripcionF1.config(text="""Se encarga de registrar cada una de las ventas, generando la factura al cliente
    self.freameCambianteFacturacion.destroy()

    self.freameCambianteFacturacion = tk.Frame(self.framePrincipal, height=150)
    self.freameCambianteFacturacion.grid(row=2, column=0, sticky="nswe")

    self.datosEntradasFacturacion=FieldFrame(self.freameCambianteFacturacion, "Detalles Venta" ,
    ["Fecha","Cliente", "sede", "Vendedor", "Empleado caja","Prenda", "Cantidad"], "valor",
    [{"Dia: {Main.fecha.getDia()}, Mes: {Main.fecha.getMes()}, Año: {Main.fecha.getAño()}"}, "", "Sede Principal", "", "",
    "", "Camisa/Pantalón", "0"], [False, True, True, False, True, True], ancho_entry=25, tamañoFuente=10)
    self.datosEntradasFacturacion.configurarCallBack("sede", "<Return>", self.actualizarDatosEmpleadosFacturacion)
    self.datosEntradasFacturacion.grid(row=1, column=0, columnspan=2)

```

En el método interaccion1Facturacion de la clase startFrame se puede encontrar otro uso del FieldFrame para que el usuario pueda llenar los datos de la compra como lo es el nombre del Cliente, la Sede en la que se encuentra, el Vendedor que lo atendió, el Empleado de Caja que registra su compra, el tipo de la Prenda comprada y la Cantidad. Además genera una entrada inhabilitada (“disabled”) donde se muestra la fecha de la compra.

```

def Interaccion2(self):
    frame2.destroy()
    frame3.destroy()

    self.fidelidadclientes = tk.Frame(framePrincipal)
    self.fidelidadclientes.pack(anchor="s", expand=True, fill="both")

    criterios = ["Fidelidad"]
    valores = ["0% / 100%"]
    habilitado = [True]

    # Creamos el FieldFrame con los botones
    field_frame2 = FieldFrame(self.fidelidadclientes, "Ingrese porcentaje a modificar para", criterios, "los clientes sin",
    field_frame2.place(relx=1, rely=0.7, relwidth=1, relheight=1, anchor="e")

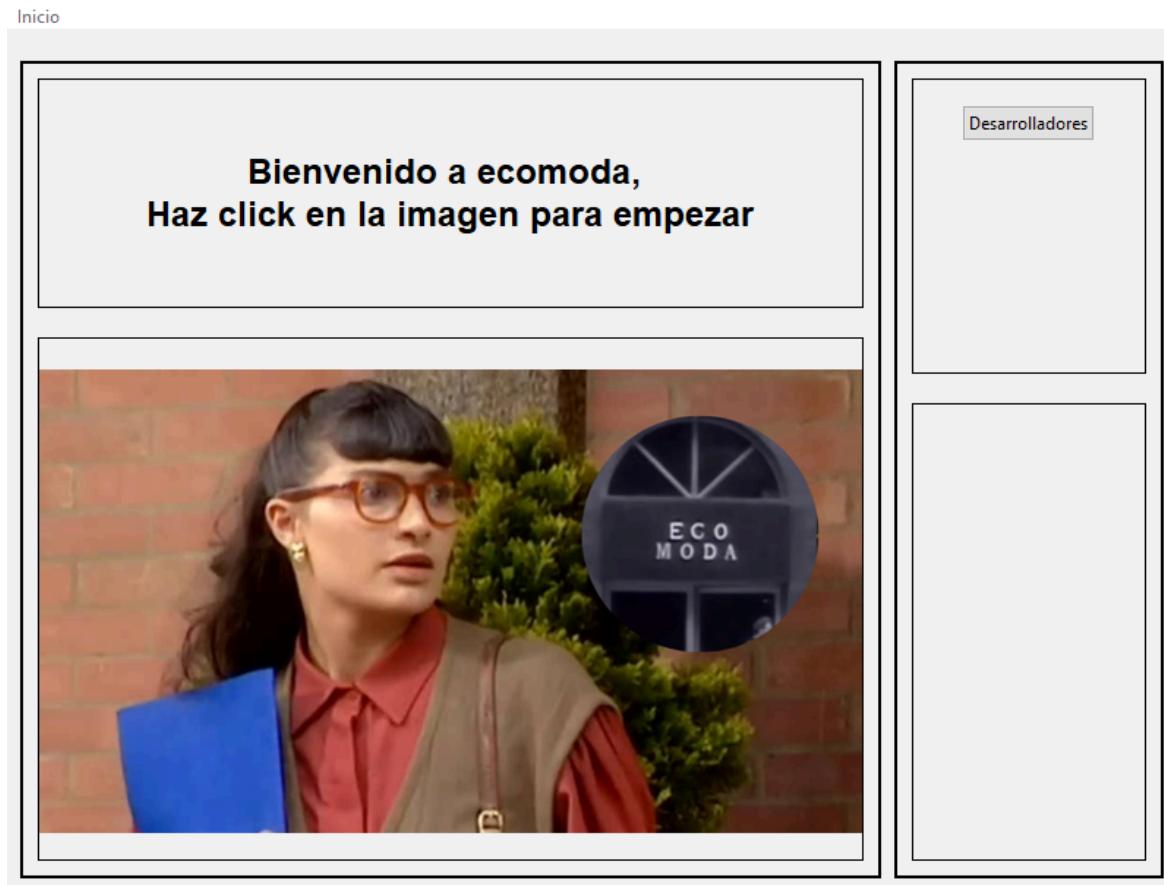
```

En el método Interaccion2 de la clase startFrame se puede encontrar otro uso del FieldFrame para que el usuario pueda ingresar el porcentaje de Fidelidad con respecto a los clientes sin membresía para tomar en cuenta al evaluar la posibilidad de aplicar descuentos en las ventas.

7. Manual de usuario

Ventana de Bienvenida

El sistema inicia en la ventana de bienvenida, que luce de la siguiente manera



En esta ventana se encuentra un menú de Inicio en la parte superior, que cuenta con las opciones de:

- Salida: Para cerrar la ventana
- Descripción: Para obtener una breve resumen de lo que se encontrara en el sistema
-

También, en la parte de la derecha se encuentra el botón “Desarrolladores” con el cual se mostrarán en el mismo espacio los integrantes del grupo con fotos y detalles de cada uno, se podrá pasar de un integrante a otro dando click sobre el recuadro que presenta el nombre y el texto personalizado para cada uno.

En la parte posterior se encuentra una serie de imágenes que cambian al pasar el cursor por ellas y al darle click se pasará a la ventana de inicio.

Ventana de Inicio

A continuación se puede apreciar la ventana, esta cuenta con un menú superior con opciones como:

- Archivo: Tiene la opción de Salir, con la cual se regresa a la ventana de bienvenida
- Procesos y Consultas: Presentan todas las funcionalidades del sistema
- Ayuda: Tiene la opción Acerca de, la cual muestra una ventana emergente con los nombres de los desarrolladores



En la parte inferior se encuentra un recuadro que contiene el espacio para que el usuario ingrese la fecha en el formato día - mes - año, este espacio contiene ciertas excepciones para que en caso de que no se ingrese una fecha válida se le notifique al usuario que es un requisito para continuar con el normal funcionamiento del sistema, con lo cual, después de ingresar una fecha correcta y presionar el botón de enviar se puede seguir al menú procesos y consultas en el cual se puede escoger la primera opción que es: *Despedir y reemplazar empleados*.

Es importante mencionar que la correcta escritura de los nombres en los campos de entrada es esencial para el correcto funcionamiento del sistema, se deben tener en cuenta las tildes.

Procesos y Consultas → Despedir y reemplazar empleados

Gestión Humana

Este área analiza la lista de todos los empleados y permite modificarla: Se puede contratar a un nuevo empleado, establecer su salario y el rol o las funciones que cumple en la empresa. También se puede despedir a un empleado ya existente en el equipo de trabajo. Con ese fin, analizamos el rendimiento de los empleados de la empresa, y llegamos a la siguiente lista de empleados insuficientes, estos pudieron ser cambiados de area o sede, y si están marcados con ¿despedir?, podrá elegirlos para despedirlos en la siguiente pantalla.

Nombre	Area	Rendimiento	Rendimiento esperado	Acción
Wilson Sastoque	Corte	0	90	Trasladado a otra area
Gabriela Garza	Corte	0	89	Trasladado a otra area
Patricia Fernandez	Ventas	17	484077	Trasladado a otra area
Miguel Robles	Oficina	0	0	Transferido a otra sede
Karina Larson	Corte	0	89	¿Despedir?
Gustavo Olarte	Direccion	20	60	¿Despedir?

Siguiente

Sobre la misma ventana de inicio aparece la lista de empleados, con su área correspondiente, rendimiento, rendimiento deseado y finalmente, si deberían ser despedidos o transferidos. Al presionar el botón Siguiente lo que hace es mostrar lo que aparece en la imagen

Gestión Humana

Los empleados de la derecha no rinden correctamente y no pudieron ser cambiados ni de area ni de sede. . .
También puede añadir a otros empleados, para buscar mas empleados, haga click en "Añadir empleado a la lista guía"
Tus empleados estan 1,542,250.0 bajo el promedio de salarios

Dato	valor
Cantidad de despedidos	2
Nombre del despedido 1	Karina Larson Gustavo Olarte
Nombre del despedido 2	

Aceptar **Borrar**

Añadir empleado a la lista guía

En el campo de cantidad de despedidos se puede cambiar el número de empleados a despedir, digitando un número entero y presionando Enter para que se habilite la cantidad dada de los campos para ingresar el nombre del empleado, ingresando el nombre completo siguiendo con la correcta escritura de cada uno de los nombres lo cual se puede verificar en la parte derecha donde se presenta la lista guía de empleados elegibles a despedir, cabe aclarar que el usuario puede despedir a cualquier empleado de la empresa, esta lista simplemente hace una sugerencia de aquellos empleados que cumplen los requerimientos para ser transferidos o despedidos, por otra parte, en el botón añadir empleado a la lista guía se presenta lo siguiente

The screenshot shows a software interface with a purple header bar containing the title "Gestión Humana". Below the header, a message box displays the instruction: "Inserte los datos de el empleado a añadir a la lista, el panel de la derecha le ayudará, presione Enter al terminar de escribir un valor". The main area contains a table with two columns: "Dato del empleado" and "valor". There are two rows: one for "sede" where the value "Sede Principal" is entered into a redacted field, and one for "nombre" with an empty input field. To the right of the table, a list of "Posibles sedes" is shown, including "Sede Principal" and "Sede 2". At the bottom of the window are two buttons: "Aceptar" (Accept) and "Borrar" (Delete).

Aquí, se debe ingresar el nombre completo de la sede con lo cual se mostrará la lista de empleados que trabaja en esa sede, en el campo de nombre se habilitará posteriormente a esa acción y en él se deberá escribir el nombre completo del empleado, al darle aceptar este empleado aparecerá en la lista guía.

Una vez seleccionados los empleados que se van a despedir se presiona el botón aceptar con lo cual se pasa a la siguiente acción que es contratar o transferir empleados para reemplazar a los que fueron despedidos.

Gestión Humana

Se han despedido 1 empleados, verificamos si se pueden reemplazar con gente de otras sedes

Nececitamos reemplazar 1 de MODISTA
Por medio de transferencia desde la sede Sede Principal.
He aquí los candidatos, escriba el nombre del seleccionado en cada casilla:
Inez Ramirez con 2 años de experiencia y 0 de pericia
Sandra Patiño con 5 años de experiencia y 0 de pericia
Sofia Lopez con 6 años de experiencia y 0 de pericia
Mariana Valdez con 10 años de experiencia y 0 de pericia
Bertha Muñoz con 15 años de experiencia y 0 de pericia

Reemplazo numero	Nombre
Reemplazo 1	

Aceptar **Borrar**

Para este proceso se tiene en cuenta el área del empleado despedido y se analizan las diferentes opciones de empleados de los que el usuario debe escoger alguno y debe ingresar el nombre completo en el campo asignado, al finalizar esta operación aparecerá un mensaje de que se han reemplazado los empleados con éxito, culminando con la funcionalidad de gestión humana, para volver a la ventana inicial se puede acceder a la opción Salir desde el menú Archivo.

Procesos y Consultas → Pedir insumos

Ahora, no es necesario volver a ingresar la fecha ya que el sistema ya la guardó, con esto, se puede acceder nuevamente a Procesos y consultas escogiendo la opción de pedir insumos con lo que aparecerá lo siguiente

Surtir Insumos

Registra la llegada de nuevos insumos: Incluye una predicción de ventas del siguiente mes para hacer la compra de los insumos, actualiza la deuda con los proveedores y añade los nuevos insumos a la cantidad en Stock.

Puede cambiar la predicción de ventas para el siguiente mes El porcentaje de pesimismo es de

Sede Principal	<input type="text" value="2"/>
Sede 2	<input type="text" value="2"/>
Aceptar	Borrar

En los campos habilitados se debe completar con un número entero que tomará el porcentaje de pesimismo que se usará para la proyección de ventas del siguiente mes o se puede dejar el número que aparezca si así se desea. Al presionar el botón aceptar aparecerá dicha predicción para cada sede, con lo cual se presiona el botón siguiente para continuar con la interacción que , en la cual se muestra por cada sede, todos los insumos con sus respectivas cantidades en bodega, cantidades necesarias y cantidades restantes o a conseguir, así

Surtir Insumos

Esta tabla le muestra los insumos necesarios para producir, y de donde se pueden sacar.

Insumo	Cantidad en bodega	Cantidad necesaria	Cantidad a conseguir	Modo para conseguir
Tela	2000	946386	944386	Comprar o transferir
Boton	80000	12428	0	¡Lo hay!
Cremallera	20	3193	3173	Comprar o transferir
Hilo	2000000	1234889	0	¡Lo hay!

Elegir sobre transferencias

y en la entrada se habilita “T/C” para que el usuario escoja si desea comprar o transferir dicho insumo si este se encuentra en otra sede, de lo contrario pasa directamente a la

lista de insumos a comprar, con el botón aceptar se muestra lo mismo para la siguiente sede,

Procesos y Consultas → Ver el desglose económico de la empresa

En procesos y consultas si se elige la opción ver el desglose económico de la empresa aparecerá

Gestión Financiera

Se realiza una evaluación del estado financiero de la empresa haciendo el cálculo de los activos y los pasivos, para indicarle al usuario qué tan bien administrada está, mostrandole los resultados y su significado

Desea calcular las	siguientes Deudas
Proveedor	<input type="checkbox"/> Si/No
Banco	<input type="checkbox"/> Si/No
<div style="border: 1px solid #ccc; width: 100%; height: 20px; margin-bottom: 10px;"></div> <div style="display: flex; justify-content: space-around;">AceptarBorrar</div>	
Calculando la diferencia entre ingresos por venta y costos de producción...	

En esta parte se debe escoger si o no, según si se quiere que el programa calcule solo las deudas con proveedores, con bancos o ambas, y abajo se encuentra un comboBox desplegable en el que se deberá seleccionar al directivo al que se le asignará el balance general. Al presionar el botón de aceptar aparece en la parte inferior de la ventana el monto del balance y se habilita el botón siguiente, con este, se pregunta por el porcentaje que se desea suponer con respecto a los clientes sin membresía que tiene que ser un número entero entre 0 y 100. Una vez ingresado, al presionar el botón de aceptar aparece en la parte inferior de la ventana el estimado de la diferencia entre ventas y deudas futuras y se habilita el botón siguiente para continuar con el proceso.



Se muestra entonces una tabla de bancos donde el usuario debe elegir uno de estos y copiar su nombre exactamente como se muestra en la tabla. Según el resultado de los cálculos de la interacción anterior, se mostrará en la parte de abajo de la ventana las deudas pagadas o la nueva deuda adquirida. Al dar click en el botón aceptar, si el nombre del banco está correctamente digitado aparecerá un botón con la palabra siguiente para poder continuar.



En este frame se le muestra al usuario el descuento recomendado por el programa según las proyecciones y se le da la opción de cambiarlo o dejarlo como está, si decide cambiarlo deberá ingresar un número positivo. Al darle aceptar se mostrará en la parte de abajo de la ventana el análisis sobre el black friday, el descuento propuesto y el descuento usado.



Finalmente, al presionar en la opción siguiente que se habilita, se muestra un resumen de los cálculos y transacciones realizados. Para volver a la ventana inicial se puede acceder a la opción Salir desde el menú Archivo.

Procesos y Consultas → Facturación

Ahora, no es necesario volver a ingresar la fecha ya que el sistema ya la guardó, con esto, se puede acceder nuevamente a Procesos y consultas escogiendo la opción de facturación con lo que aparecerá lo siguiente:

Facturacion

Se encarga de registrar cada una de las ventas, generando la factura al cliente con los datos necesarios.
Inserte los datos de la sede y presione Enter para ver los empleados

Detalles Venta	valor
Fecha	Dia: 25 Mes: 1 Año: 25
Cliente	<input type="text"/>
sede	Sede Principal
Vendedor	<input type="text"/>
Empleado caja	<input type="text"/>
Prenda	Camisa/Pantalon
Cantidad	0

Clientes
Claudia Elena Vásquez
Michel Doniel
Claudia Bosch
Mónica Agudelo
Daniel Valencia
Efraín Rodríguez
Mauricio Brightman
Nicolás Mora
Roberto Mendoza
Hermes Pinzón
Julia Solano
Maria Beatriz Valencia
Antonio Sánchez
Armando Paredes
Nuria Rendón

Posibles sedes:
Sede Principal
Sede 2

Aceptar **Borrar**

Inicialmente, se debe llenar el campo de Cliente con uno de los nombres que aparece en la columna de clientes y el campo de Sede debe llenarse con una de las sedes que aparece bajo Posibles Sedes. Una vez llenos estos campos, se debe dar enter en la entrada de la sede para que el programa muestre los empleados disponibles y habilite los campos.

Ahora, en la casilla de vendedor se debe poner el nombre de uno de los empleados que se muestran bajo el texto Vendedores posibles, y para la casilla de Empleado caja se debe poner el nombre de uno de los empleados que se muestran bajo el texto Empleados de caja posibles, respetando tildes y mayúsculas en ambos casos. Para Prenda se debe llenar con la palabra Camisa o Pantalón, cualquier otra prenda será inválida, y cantidad se deberá llenar con un número entero mayor que cero. Una vez terminada, al presionar en el botón aceptar se lanzará una excepción preguntando si desea añadir más prendas, si responde si, se bloquearan todos los campos excepto Prenda y Cantidad para que añada la prenda que desee y su respectiva cantidad (puede ser la misma prenda que antes pero se sumará la cantidad).

Facturacion

Se encarga de registrar cada una de las ventas, generando la factura al cliente con los datos necesarios.
Inserte los datos de la sede y presione Enter para ver los empleados

Detalles Venta	valor
Fecha	Dia: 25 Mes: 1 Año: 25
Cliente	Michel Doniel
sede	Sede Principal
Vendedor	Catalina Ángel
Empleado caja	Aura Maria
Prenda	Pantalon
Cantidad	1

<p>Cuentas</p> <p>Claudia Elena Vásquez Michel Doniel Claudia Bosch Mónica Agudelo Daniel Valencia Efraín Rodríguez Mauricio Brightman Nicolás Mora Roberto Mendoza Hermes Pinzón Julia Solano Maria Beatriz Valencia Antonio Sánchez Armando Paredes Nuria Rendón</p>	<p>Vendedores posibles</p> <p>Catalina Ángel Mario Calderón</p> <p>Empleados de caja posibles</p> <p>Aura Maria Wilson Sastogue</p>
--	---

Cuando decida que ha terminado la selección de prendas, seleccione la opción de No cuando se lance la excepción y presione el botón Siguiente que aparece en pantalla. Se le mostrarán entonces tres entradas, cada una representa la cantidad a pedir de un tipo de bolsa, el usuario debe ingresar números enteros positivos entre cero y el máximo de bolsas que se le muestra en la parte de abajo. Luego de ingresar los datos debe presionar el botón aceptar.

Este proceso se hará hasta que termine de empacar sus prendas en las bolsas. Cuando esto suceda se bloquearán las entradas y se dará la opción de siguiente.

Facturacion

Se encarga de seleccionar bolsas para la compra.

Tamaño bolsa	Bolsas Necesarias
Grande	0
Medianas	0
Pequeña	1

Hay máximo 20 bolsas grandes, 20 bolsas medianas y 20 bolsas pequeñas.

Luego, se le pregunta al usuario cuántas bolsas desea comprar, debe llenar el campo con un número entero mayor o igual a cero, presionar aceptar y luego siguiente.

Facturacion

Se encarga de surtir bolsas de ser necesario.

Cantidad Bolsas Cantidad que desea Comprar

Cantidad a comprar

5

Aceptar

Borrar

No hay necesidad de comprar

Acto seguido, se mostrarán las entradas Código, Nueva tarjeta y Monto nueva Tarjeta. En código deberá poner un código de regalo válido o -1 en caso de no tener o no querer utilizar el código, en nueva tarjeta deberá ingresar si o no según quiera comprar una nueva tarjeta y en monto nueva tarjeta, el monto que esta tendría. Cuando termine de llenar los datos seleccione aceptar y luego siguiente.

Facturacion

Se encarga de Redimir y/o comprar tarjetas de regalo.
Ingrese -1 si no desea redimir ninguna tarjeta y Si si no desea comprar.

Targeta de regalo

Código

-1

Nueva tarjeta

Si/No

Monto nueva Tarjeta

100000

Aceptar

Borrar

Aparecerán entonces dos entradas, primero, debe responder si o no a la primera entrada según quiera o no transferir los fondos a la cuenta principal. Al haber llenado estos datos podemos dar enter encima de dicha entrada y se habilitará la opción del porcentaje que desea transferir para que el usuario ingrese un número entre 20 y 60. Finalmente debe dar click en aceptar y en siguiente.

Facturacion

Se encarga de transferir los fondos a la cuenta principal.

Fondos

Transferir fondos a la cuenta principal

¿Qué porcentaje desea transferir? 20% o 60%

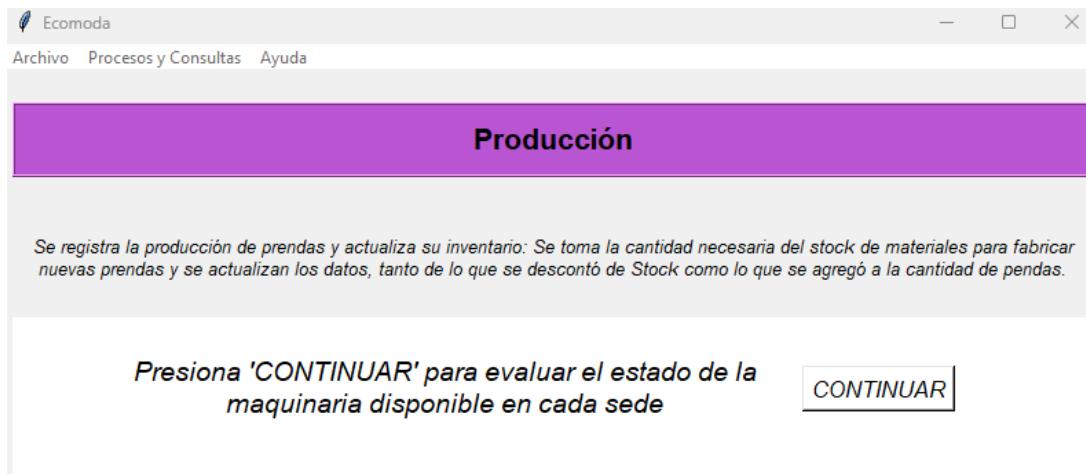
Aceptar

Borrar

Al final se mostrará en pantalla la factura de la compra y el valor almacenado en la cuenta de la sede. Para volver a la ventana inicial se puede acceder a la opción Salir desde el menú Archivo.

Procesos y Consultas → Producir Prendas

Ahora, no es necesario volver a ingresar la fecha ya que el sistema ya la guardó, con esto, se puede acceder nuevamente a Procesos y consultas escogiendo la opción de producir prendas con lo que aparecerá lo siguiente, dónde deberá presionar el botón continuar para que el programa comience a analizar la información del sistema.



Se le mostrará en pantalla el proveedor más barato del repuesto a surtir, el nombre del repuesto, la máquina afectada y la sede de la que se extraerá el dinero. Deberá llevar la entrada Sede para pagar con el nombre de una de las sedes válidas (Sede Principal o Sede 2). Cuando termine presione aceptar. **Todo este proceso se repite hasta que se terminen de surtir todos los repuestos necesarios.**



Cuando se terminen de surtir todos repuestos se podrá acceder al informe de los repuestos comprados y las máquinas inhabilitadas por falta de revisión (mantenimiento). Debe pulsar Maquinaria Disponible para seguir a la siguiente pantalla.



Se muestran en pantalla las máquinas de cada una de las sedes junto con las horas de uso de cada una de ellas, acompañado de un texto que indica cuál o cuáles sedes están disponibles para la producción. Para continuar presione Planificar Producción,



Se mostrará una pantalla con la producción actual y la que queda pendiente para cada sede y tipo de prenda. Si el usuario desea modificar la cantidad a producir, por ejemplo de

pantalones, debe hacer click en Modificar Pantalones y llevar la flecha hasta el valor que desea modificar, luego, deberá ingresar el valor a modificar en la entrada que dice modificar aquí y dar enter para que se actualicen los datos. Una vez esté de acuerdo con los valores presione continuar. Se lanzará una excepción con los valores de sobrecosto de cada sede y pregunta si desea continuar.



Se muestran los datos de los insumos que hay en cada Sede y una lista de modistas. En esta pantalla se debe presionar el botón referente al modista que guste seleccionar y repetir este proceso hasta que se acaben las tandas de producción.



Se muestran las prendas producidas en cada sede y las que quedaron pendientes. Para salir seleccione la opción del menú archivo, salir.

Archivo Procesos y Consultas Ayuda

Producción

Se registra la producción de prendas y actualiza su inventario: Se toma la cantidad necesaria del stock de materiales para fabricar nuevas prendas y se actualizan los datos, tanto de lo que se descontó de Stock como lo que se agregó a la cantidad de prendas.

PRODUCCIÓN SALIENTE

Sede Principal

Pantalones: 10
Camisas: 0

Sede 2

Pantalones: 79
Camisas: 0

PRODUCCIÓN LANZADA A MAÑANA

Sede Principal

Pantalones: 0
Camisas: 0

Sede 2

Pantalones: 0
Camisas: 0

No se pudo producir todo porque los insumos no alcanzaron, producimos 89 prendas