Informe Integral de Ingeniería de Software: Paradigmas de Diseño Orientado a Objetos en Python

1. Introducción al Paradigma de Objetos en la Ingeniería Moderna

La Programación Orientada a Objetos (POO) constituye la piedra angular del desarrollo de software contemporáneo, proporcionando un marco conceptual que permite modelar sistemas complejos a través de la interacción de entidades autónomas denominadas objetos. En el ecosistema del lenguaje Python, la POO no es simplemente una característica opcional, sino un rasgo intrínseco de su arquitectura; desde los tipos de datos primitivos hasta las funciones y módulos, todo en Python es un objeto. Sin embargo, la verdadera potencia de este paradigma no reside en la mera instanciación de clases aisladas, sino en la orquestación sofisticada de sus interacciones.

El presente informe técnico ofrece un análisis exhaustivo de los patrones estructurales y de comportamiento que gobiernan estas interacciones, con un enfoque específico en los mecanismos de **Colaboración** y **Composición**. A diferencia de la herencia tradicional, que establece jerarquías rígidas y a menudo frágiles, la composición y la colaboración permiten construir sistemas flexibles, modulares y mantenibles. A lo largo de este documento, desglosaremos la mecánica interna del diseño de clases en Python, abordando la gestión de estado a través de constructores, la encapsulación mediante accesadores y mutadores, y las estrategias para implementar polimorfismo y sobrecarga en un lenguaje de tipado dinámico.

El objetivo final es sintetizar estos conceptos teóricos en una solución de software práctica y robusta: un sistema de facturación de ventas. Este problema, aunque de baja complejidad algorítmica, sirve como el escenario ideal para demostrar cómo las decisiones arquitectónicas sobre las relaciones entre objetos afectan directamente la calidad, legibilidad y escalabilidad del código.[1]

2. Marco Conceptual: Taxonomía de las Relaciones entre Objetos (2.1)

Para resolver problemas computacionales, los objetos deben comunicarse. Esta comunicación se define por las relaciones establecidas entre las clases. En la teoría estricta de la POO, y particularmente en el modelado UML (Lenguaje Unificado de Modelado), estas relaciones se categorizan por la fuerza de la conexión y la dependencia del ciclo de vida entre las entidades involucradas.[3]

2.1.1 La Jerarquía de las Asociaciones

En el nivel más alto de abstracción, cualquier interacción entre dos clases se denomina **Asociación**. Una asociación sugiere que un objeto "conoce" a otro o tiene una referencia a él. Sin embargo, para diseñar software efectivo, es imperativo distinguir entre los matices de estas asociaciones, ya que cada una implica diferentes compromisos de memoria y mantenimiento.

La siguiente tabla resume las diferencias críticas entre los tipos de relaciones que exploraremos, proporcionando una guía rápida para la toma de decisiones arquitectónicas:

Tipo de Relación	Semántica	Ciclo de Vida	Acoplamiento	Ejemplo Canónico
Asociación	"Conoce a"	Independiente	Bajo	Un Estudiante usa un Libro.
Colaboración	"Usa a" / "Coopera con"	Transitorio (método)	Muy Bajo	Un Conductor usa un GPS para una ruta.
Agregación	"Tiene un" (Débil)	Independiente	Medio	Un Departamento tiene Profesores.
Composición	"Contiene un" (Fuerte)	Dependiente / Vinculado	Alto	Una Casa tiene Habitaciones.

Colaboración (La Relación "Usa-A")

La colaboración se describe a menudo como la forma más transitoria de asociación. En una relación colaborativa, dos objetos trabajan juntos para realizar una tarea específica, pero permanecen como entidades fundamentalmente independientes.[1] No existe una relación de propiedad a largo plazo.

- **Mecanismo**: El Objeto A solicita un servicio del Objeto B. Típicamente, el objeto colaborador se pasa como argumento a un método del objeto cliente.
- **Ciclo de Vida**: La relación puede existir solo durante la ejecución de una sola llamada a función. Una vez que la función retorna, el objeto cliente puede "olvidar" al colaborador.
- **Implicaciones en Python**: La colaboración es intrínseca al concepto de **Inyección de Dependencias**. En lugar de que una clase codifique rígidamente sus utilidades (como una conexión a base de datos o un servicio de impuestos), recibe estos colaboradores desde el exterior. Esto facilita enormemente las pruebas unitarias (testing), ya que los colaboradores reales pueden ser sustituidos por "mocks" o simulacros.[6]

Un ejemplo clásico de colaboración en Python se observa cuando un objeto de negocio delega una tarea de cálculo a un objeto de servicio. El objeto de negocio no necesita saber *cómo* se realiza el cálculo, solo que el colaborador ofrece esa funcionalidad.

Agregación (La Relación "Tiene-A" Débil)

La agregación representa una relación de "todo-parte" donde el objeto hijo puede existir independientemente del padre. Se refiere a menudo como una asociación "débil" porque la destrucción del contenedor no implica la destrucción del contenido.[3]

- **Metáfora Visual**: Un `Aeropuerto` y sus `Aviones`. El aeropuerto "tiene" aviones, pero si el aeropuerto cierra, los aviones no son desguazados; simplemente vuelan a otro lugar.
- **Implementación en Python**: El objeto padre mantiene una referencia al objeto hijo (generalmente en una lista o variable de instancia), pero el objeto hijo se instancia fuera del padre y se pasa a través del constructor o un método de adición (`add_plane`).

Composición (La Relación "Tiene-A" Fuerte)

La composición es la forma más estricta de agregación y el foco central del desafío de diseño de este informe. Implica propiedad exclusiva y un ciclo de vida vinculado. Si el objeto padre es destruido, los objetos hijos dentro de él dejan de existir lógicamente dentro del contexto de ese sistema.[2]

- **Metáfora Visual**: Un `Coche` y su `Motor`. En el contexto de un modelo de datos de fábrica, el motor se crea específicamente para ese coche. Si el objeto `Coche` es eliminado de la memoria, el `Motor` asociado también lo es, ya que no tiene propósito sin el chasis que lo contiene.
- **Implementación en Python**: Los objetos hijos se instancian típicamente *dentro* del constructor (`__init__`) de la clase padre. Esto asegura que el padre controle la creación y destrucción del hijo.

El Debate: Composición sobre Herencia

Un principio recurrente en la ingeniería de software moderna es "Favorecer la composición sobre la herencia".2 La herencia ("Es-un") crea estructuras fuertemente acopladas donde una subclase depende de los detalles de implementación del padre. Esto puede llevar al problema de la "clase base frágil", donde cambios en el padre rompen la funcionalidad del hijo. La composición ("Tiene-un") permite un diseño modular donde el comportamiento se ensambla a partir de componentes más pequeños e independientes. Esto permite que una clase cambie su comportamiento en tiempo de ejecución intercambiando componentes, una flexibilidad que las jerarquías de herencia estática no pueden proporcionar.

En Python, la gestión de memoria para la composición es manejada automáticamente por el recolector de basura (Garbage Collector). Cuando el objeto contenedor pierde todas sus referencias y es recolectado, los objetos contenidos (si no tienen otras referencias externas) también son recolectados, cumpliendo así con la semántica de ciclo de vida de la composición.[2]

2.1.2 Navegando la Dicotomía "Colaboración vs. Composición" en Python

La distinción se vuelve crítica al definir los límites de un módulo o microservicio.

- **La Colaboración** trata sobre el **comportamiento**. Responde a la pregunta: "¿Quién me ayuda a hacer esto?".
- **La Composición** trata sobre la **estructura**. Responde a la pregunta: "¿De qué estoy hecho?".

En el contexto del programa solicitado (un sistema de facturación de baja complejidad), utilizaremos **Composición** para construir la estructura de datos primaria (una Factura está compuesta de Líneas de Detalle) y **Colaboración** para manejar operaciones externas (la Factura colabora con una calculadora de impuestos o un formateador de impresión). Esto demuestra una comprensión matizada de que las aplicaciones del mundo real rara vez usan un solo patrón; son un tapiz de relaciones de fuerza variable.

3. Mecánica Avanzada de Clases: Constructores, Accesadores y Mutadores (2.2)

Una vez definidas las relaciones estructurales, la integridad interna de cada clase debe ser asegurada. Esto se logra a través de la encapsulación: ocultar el estado interno y exponer una interfaz pública controlada. En Python, esto implica una comprensión profunda de cómo el lenguaje maneja la visibilidad de atributos y la intercepción de accesos.

3.1 Constructores: La Puerta de Entrada al Estado (`__init__`)

En Python, la inicialización de un objeto es manejada predominantemente por el método `__init__`. Aunque a menudo se le llama coloquialmente "constructor", técnicamente Python separa la creación de la memoria (`__new__`) de la inicialización (`__init__`). Para la inmensa mayoría de la programación de aplicaciones, `__init__` es el foco principal.[11]

El método `__init__` es el primer punto donde se aplica la **Composición**. Si la Clase A está compuesta por la Clase B, el método `__init__` de la Clase A es responsable de instanciar y configurar la Clase B.

```
Python

class Motor:
    def arrancar(self):
        print("Motor en marcha")

class Automovil:
    def __init__(self):
        # Composición: El Automóvil crea su propio Motor.
        # El Motor es una parte integral del Automóvil.
        self.motor = Motor()
```

Una consideración vital en los constructores de Python es el manejo de argumentos por defecto mutables (como listas o diccionarios). Definir `def __init__(self, items=)` es un error común que lleva a que todas las instancias de la clase compartan la misma lista. La práctica correcta para la composición es usar `None` y crear la estructura dentro del método:

```
Python

def __init__(self, items=None):
    if items is None:
        self.items = # Nueva lista para cada instancia
    else:
        self.items = items
```

3.2 Encapsulación y Control de Acceso: La Filosofía Pythonica

Python difiere significativamente de lenguajes como Java o C++ con respecto al control de acceso. Python opera bajo el principio de "todos somos adultos responsables" (*we are all consenting adults*), lo que significa que carece de modificadores de acceso estrictos como `private` o `protected` impuestos por el compilador o el intérprete en tiempo de ejecución.[13] En su lugar, se basa en convenciones de nomenclatura fuertemente respetadas por la comunidad:

1. **Atributos Públicos (`nombre`)**: Accesibles desde cualquier lugar. Indican la interfaz pública estable de la clase.
2. **Atributos Protegidos (`_nombre`)**: Un solo guion bajo prefijado indica a otros desarrolladores (y a los linters de código) que esta variable está destinada solo para uso interno o para subclases. Técnicamente es accesible, pero acceder a ella desde fuera se considera una mala práctica.
3. **Atributos Privados (`__nombre`)**: Un doble guion bajo activa el "name mangling" (mutilación de nombres). El intérprete de Python reescribe internamente el atributo como `_NombreClase__nombre`, haciéndolo más difícil (aunque no imposible) de acceder accidentalmente. Su propósito principal es evitar conflictos de nombres en jerarquías de herencia complejas, no la seguridad estricta.[14]

3.3 La Evolución de Accesadores y Mutadores

En la POO tradicional (estilo Java), el acceso directo a los atributos está prohibido. Los desarrolladores deben escribir métodos explícitos Getter y Setter para cada variable.

Java

```
// Estilo Java (No Pythonico)
public class Persona {
    private String nombre;
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
}
```

La adopción temprana de Python a menudo imitaba este patrón, resultando en código verboso y poco "Pythonico".[13] Escribir getters y setters triviales (accesadores y

mutadores) para cada atributo satura la base de código y no ofrece ningún beneficio inicial si no hay lógica adjunta al acceso.

3.3.1 La Vía Pythonica: El Decorador `@property`

Python resuelve el dilema de la encapsulación con el decorador `@property`. Esta característica permite a los desarrolladores comenzar con atributos públicos (ej. `obj.precio`) y, si se necesita lógica de validación más tarde, convertirlos en propiedades sin romper la API externa (la sintaxis `obj.precio` sigue siendo la misma, pero ahora activa un método tras bambalinas).[13]

Esta capacidad se basa en el **Protocolo Descriptor** de Python, un mecanismo avanzado que permite personalizar cómo se recuperan y establecen los atributos.

Los Componentes de una Propiedad:

1. **El Getter (`@property`)**: Un método decorado que permite acceder al atributo como si fuera una variable. Este es el **Accesador**. Se utiliza para devolver valores privados o para calcular valores al vuelo (propiedades computadas).
2. **El Setter (`@nombre.setter`)**: Un método que define cómo se debe asignar el valor. Este es el **Mutador**. Es el lugar ideal para la validación de entrada (por ejemplo, asegurar que un precio no sea negativo) antes de que el valor se almacene en el atributo privado subyacente (`_precio`).
3. **El Deleter (`@nombre.deleter`)**: Maneja la lógica necesaria cuando se elimina el atributo con `del`.

Mejores Prácticas y Toma de Decisiones:

Utilizar atributos públicos para datos simples es aceptable y fomentado. Se debe migrar a @property cuando se necesite controlar el acceso (hacer atributos de solo lectura eliminando el setter), validar la entrada (lógica en el mutador), o computar un valor dinámicamente basado en otros atributos.13

En la solución de este reporte, utilizaremos `@property` extensivamente para garantizar la integridad de los datos financieros, asegurando que el problema de "baja complejidad" se resuelva con estándares de calidad de producción, evitando estados inválidos como precios negativos o cantidades cero.

4. Polimorfismo y Sobrecarga en un Entorno Dinámico (2.3)

Uno de los puntos de confusión más comunes para los desarrolladores que transicionan a Python desde lenguajes estáticamente tipados es el concepto de **Sobrecarga** (Overloading).

4.1 El Mito de la Sobrecarga Nativa

La **Sobrecarga de Métodos** tradicional implica definir múltiples métodos con el *mismo nombre* pero *diferentes firmas* (diferentes tipos o cantidades de parámetros). En C++ o Java, el compilador decide qué método llamar basándose en los argumentos pasados en tiempo de compilación.

```
Java

// Ejemplo en Java (Sobrecarga)

void imprimir(String s) {... }

void imprimir(int i) {... }
```

Python **no soporta sobrecarga de métodos nativa** en este sentido estricto. Dado que Python es dinámicamente tipado y los nombres de funciones son esencialmente claves en un diccionario de espacio de nombres (`__dict__`), definir un método `func(x)` y luego `func(x, y)` simplemente resultará en que la segunda definición sobrescriba a la primera. La primera versión deja de existir.[11]

4.2 Estrategias de Simulación de Sobrecarga

A pesar de la falta de soporte nativo, Python proporciona mecanismos poderosos para simular el comportamiento de sobrecarga, ofreciendo una flexibilidad que a menudo supera a la sobrecarga estática.

4.2.1 Simulación A: Argumentos por Defecto (La Vía Estándar)

La forma más común y "Pythonica" de manejar números variables de argumentos es a través de **Parámetros por Defecto**.[20]

```
Python

def conectar(self, host, puerto=80, timeout=30):
    # Lógica de conexión
    pass
```

Este único método puede manejar llamadas con 1, 2 o 3 argumentos, cubriendo eficazmente los casos de uso de tres métodos sobrecargados separados en Java, pero con una sintaxis más limpia.

4.2.2 Simulación B: Argumentos Variables (*args, **kwargs)

Para números indeterminados de argumentos, Python utiliza los operadores de desempaquetado *args (tupla de argumentos posicionales) y **kwargs (diccionario de argumentos de palabra clave).[22] Esto permite que un método acepte cualquier número de entradas y las procese dinámicamente iterando sobre la tupla o el diccionario.

- *args: Recoge argumentos posicionales extra. Útil para funciones matemáticas como sumar(a, b, *mas_numeros).
- **kwargs: Recoge argumentos nombrados extra. Útil para pasar opciones de configuración a colaboradores o superclases.

4.2.3 Simulación C: Despacho Múltiple (Overloading por Tipo)

Para la sobrecarga basada en **tipo** (ej. procesar(int) vs procesar(str)), los desarrolladores de Python pueden usar el decorador functools.singledispatch (biblioteca estándar) o bibliotecas de terceros como multipledispatch.[22] Esto permite registrar diferentes funciones manejadoras para diferentes tipos de argumentos, imitando el polimorfismo verdadero.

4.3 Sobrecarga de Constructores

Al igual que con los métodos, una clase solo puede tener un __init__. Para crear objetos de diferentes maneras (por ejemplo, crear un objeto Fecha desde una cadena "2023-01-01" vs. desde tres enteros 2023, 1, 1), Python utiliza **Métodos de Clase** como constructores alternativos.[26]

Patrón de Diseño:

1. __init__: El inicializador canónico que toma los argumentos más crudos o directos.
2. @classmethod: Métodos que procesan la entrada y devuelven una instancia de la clase (cls(...)). A menudo se nombran con el prefijo from_, como from_string o from_json.

Este patrón es más limpio que un __init__ monolítico lleno de lógica de comprobación de tipos if/else, promoviendo el Principio de Responsabilidad Única.

5. Implementación Práctica: Sistema de Facturación de Ventas

Para demostrar la integración de estos conceptos (Colaboración, Composición, Constructores, Accesadores/Mutadores y Sobrecarga), diseñaremos e implementaremos un **Sistema de Facturación de Ventas**.

5.1 Definición del Problema y Diseño de Clases

Requerimos un programa capaz de generar facturas detalladas. El análisis de dominio revela las siguientes entidades y sus relaciones:

1. **Producto (`Producto`)**:
    - **Responsabilidad**: Representar un ítem disponible para la venta.
    - **Atributos**: Nombre (str), Precio (float).
    - **Mecánica**: Debe validar que el precio nunca sea negativo mediante mutadores.
2. **Cliente (`Cliente`)**:
    - **Responsabilidad**: Representar al comprador.
    - **Atributos**: Nombre, ID, Email.
3. **Línea de Factura (`LineaFactura`)**:
    - **Responsabilidad**: Vincular un producto con una cantidad específica en el contexto de una venta.
    - **Relación (Asociación/Composición)**: Una Línea de Factura *tiene un* Producto. Es el bloque constructivo de la factura.
    - **Atributos**: Producto (obj), Cantidad (int), Subtotal (computado).
4. **Calculadora de Impuestos (`CalculadoraImpuestos`)**:
    - **Responsabilidad**: Proveer lógica de cálculo fiscal.
    - **Relación (Colaboración)**: La Factura *usa* la Calculadora. No es dueña de ella; es un servicio inyectado.
5. **Factura (`Factura`)**:
    - **Responsabilidad**: Agrupar líneas de venta y calcular totales.
    - **Relación (Composición)**: Una Factura *está compuesta por* una lista de Líneas de Factura. Si la factura se anula o elimina, sus líneas pierden sentido.
    - **Sobrecarga**: La Factura debe permitir agregar ítems de múltiples formas (pasando un objeto Producto existente o pasando nombre y precio "al vuelo").

5.2 Implementación del Código Fuente

A continuación se presenta la implementación completa en Python, anotada para resaltar los conceptos teóricos discutidos.

Python

```
import uuid

from datetime import datetime

from typing import List, Union, Optional



# ========================================
```

```python
# 1. SERVICIO DE COLABORACIÓN (Colaborador)
# ==========================================
class CalculadoraImpuestos:
    """
    Clase de servicio que representa un Colaborador.
    La clase Factura 'usará' esta clase para realizar cálculos, pero no la
    'poseerá' en sentido estricto (no gestiona su ciclo de vida).
    Esto permite inyectar diferentes estrategias fiscales (IVA 21%, IVA 10%, Exento).
    """
    def __init__(self, tasa: float = 0.21):
        self._tasa = tasa

    def calcular_impuesto(self, monto: float) -> float:
        """Método de servicio puro."""
        return monto * self._tasa

    @property
    def tasa_porcentual(self) -> str:
        """Accesador para representación legible."""
        return f"{self._tasa * 100:.1f}%"

# ==========================================
# 2. ENTIDADES DE DOMINIO (Encapsulación & Accesadores)
# ==========================================
class Producto:
    """
```

```
    Representa un producto en el inventario.
    Conceptos:
    - Constructores (__init__)
    - Encapsulación (Atributos protegidos)
    - Accesadores/Mutadores (@property)
    """
    def __init__(self, nombre: str, precio: float):
        self._nombre = nombre
        # Asignamos a la propiedad 'precio', NO a la variable _precio directamente.
        # Esto fuerza a que la lógica de validación del setter se ejecute
        # incluso durante la construcción del objeto.
        self.precio = precio

    # 2.2 Accesador (Getter)
    @property
    def nombre(self) -> str:
        return self._nombre

    # 2.2 Accesador (Getter)
    @property
    def precio(self) -> float:
        return self._precio

    # 2.2 Mutador (Setter) con Lógica de Validación
    @precio.setter
    def precio(self, valor: float):
```

```
        if valor < 0:
            raise ValueError(f"Error Crítico: El precio de '{self._nombre}' no puede ser negativo.")
        self._precio = valor

    def __str__(self):
        return f"{self._nombre} (${self._precio:.2f})"

class Cliente:
    """
    Entidad simple de datos para el cliente.
    """
    def __init__(self, nombre: str, email: str):
        self.id = str(uuid.uuid4())[:8] # Generación de ID único
        self.nombre = nombre
        self.email = email

    def __str__(self):
        return f"{self.nombre} ({self.email})"

# ========================================
# 3. COMPONENTES DE COMPOSICIÓN
# ========================================
class LineaFactura:
    """
    Representa una línea individual en la factura.
    Relación: LineaFactura --(Asociación)--> Producto.
```

```
    """
    def __init__(self, producto: Producto, cantidad: int):
        self._producto = producto
        self.cantidad = cantidad # Dispara validación del setter

    @property
    def cantidad(self) -> int:
        return self._cantidad

    @cantidad.setter
    def cantidad(self, valor: int):
        if valor <= 0:
            raise ValueError("La cantidad debe ser al menos 1.")
        self._cantidad = valor

    @property
    def subtotal(self) -> float:
        # Propiedad Computada: No requiere almacenamiento (setter),
        # se deriva dinámicamente del estado de los colaboradores.
        return self._producto.precio * self._cantidad

    def obtener_detalle(self) -> str:
                return f"{self._producto.nombre.ljust(20)} x {str(self.cantidad).center(4)} =
${self.subtotal:.2f}"

# ========================================
```

```
# 4. CLASE COMPUESTA PRINCIPAL (El Agregador)
# ==========================================
class Factura:
    """
    La clase central que orquesta el sistema.
    Conceptos:
    - Composición: Posee una lista interna de LineaFactura.
    - Colaboración: Utiliza CalculadoraImpuestos.
    - Sobrecarga Simulada: Método agregar_item polimórfico.
    """
    def __init__(self, cliente: Cliente, servicio_impuestos: CalculadoraImpuestos):
        self.numero_factura = f"FAC-{datetime.now().strftime('%Y%m%d')}-{str(uuid.uuid4())[:4]}"
        self.fecha_emision = datetime.now()
        self.cliente = cliente

        # Colaboración: Inyección de Dependencia
        # Guardamos la referencia al servicio que nos ayudará a calcular.
        self._servicio_impuestos = servicio_impuestos

        # Composición: La lista de ítems nace y vive dentro de la Factura.
        # Inicializamos la lista vacía. Esta lista es parte del estado interno de la Factura.
        self._items: List[LineaFactura] =

    # 2.3 Simulación de Sobrecarga (Overloading)
    # Queremos una API flexible que permita agregar ítems de dos formas:
    # 1. Pasando un objeto Producto ya creado.
```

```
    # 2. Pasando un nombre y precio para crear un producto "ad-hoc".
    def agregar_item(self, producto_o_nombre: Union[Producto, str], cantidad: int = 1, precio: float = 0.0):
        """
        Simula Sobrecarga mediante chequeo de tipos (isinstance) y argumentos por defecto.

        Firma A: agregar_item(objeto_producto, cantidad)
        Firma B: agregar_item(nombre_str, cantidad, precio)
        """
        if isinstance(producto_o_nombre, Producto):
            # Lógica para Firma A: Composición con objeto existente
            nuevo_item = LineaFactura(producto_o_nombre, cantidad)
            self._items.append(nuevo_item)
            print(f"[Log] Ítem agregado vía Objeto: {producto_o_nombre.nombre}")

        elif isinstance(producto_o_nombre, str):
            # Lógica para Firma B: Creación y Composición "al vuelo"
            # Aquí la Factura actúa como una fábrica temporal.
            if precio <= 0:
                raise ValueError("Debe proporcionar un precio positivo al agregar por nombre.")

            producto_temporal = Producto(producto_o_nombre, precio)
            nuevo_item = LineaFactura(producto_temporal, cantidad)
            self._items.append(nuevo_item)
            print(f"[Log] Ítem agregado vía Definición Directa: {producto_o_nombre}")
```

```
        else:
            raise TypeError("Tipo de argumento inválido para agregar_item")

    # Propiedades Computadas para Totales
    @property
    def total_neto(self) -> float:
        """Suma de todos los subtotales de los componentes."""
        return sum(item.subtotal for item in self._items)

    @property
    def monto_impuesto(self) -> float:
        """Uso del Colaborador para lógica externa."""
        return self._servicio_impuestos.calcular_impuesto(self.total_neto)

    @property
    def gran_total(self) -> float:
        return self.total_neto + self.monto_impuesto

    def generar_recibo(self):
        """Método de visualización."""
        print(f"\n{'='*50}")
        print(f"FACTURA: {self.numero_factura}")
        print(f"Cliente: {self.cliente}")
        print(f"Fecha:   {self.fecha_emision.strftime('%Y-%m-%d %H:%M')}")
        print(f"{'-'*50}")
        print(f"{'PRODUCTO'.ljust(20)} {'CANT'.center(4)}   {'SUBTOTAL'}")
```

```
        print(f"{'-'*50}")

        for item in self._items:
            print(f" {item.obtener_detalle()}")

        print(f"{'-'*50}")
        print(f"Neto:          ${self.total_neto:>10.2f}")
                                print(f"Impuesto      ({self._servicio_impuestos.tasa_porcentual}):
${self.monto_impuesto:>10.2f}")
        print(f"{'='*50}")
        print(f"TOTAL A PAGAR:   ${self.gran_total:>10.2f}")
        print(f"{'='*50}\n")

# ==========================================
# 5. BLOQUE DE EJECUCIÓN Y PRUEBAS
# ==========================================
if __name__ == "__main__":
    print(">>> Iniciando Sistema de Facturación...\n")

    # 1. Configuración de Colaboradores
    # Podemos cambiar fácilmente la estrategia fiscal aquí.
    impuesto_general = CalculadoraImpuestos(tasa=0.19) # 19% IVA

    # 2. Creación de Entidades Base
    cliente_vip = Cliente("María García", "maria.garcia@email.com")
```

```
laptop = Producto("Laptop Gamer", 1500.00)
mouse = Producto("Mouse Inalámbrico", 25.00)

# 3. Instanciación de la Factura (Inicio de la Composición)
factura = Factura(cliente_vip, impuesto_general)

# 4. Uso de Sobrecarga (Firma A: Objetos Producto)
factura.agregar_item(laptop, 1)
factura.agregar_item(mouse, 2)

# 5. Uso de Sobrecarga (Firma B: Datos Crudos)
# Esto demuestra flexibilidad en la API para items ad-hoc
try:
    factura.agregar_item("Servicio Instalación", 1, 50.00)
except ValueError as e:
    print(f"Error: {e}")

# 6. Generación del Reporte
factura.generar_recibo()

# 7. Demostración de Encapsulación/Validación (Mutadores)
print(">>> Prueba de Seguridad (Validación de Precios):")
try:
    laptop.precio = -100.00 # Intento de sabotaje
except ValueError as e:
    print(f"ALERTA DE SEGURIDAD CAPTURADA: {e}")
```

```
print(f"El precio se mantuvo seguro en: ${laptop.precio}")
```

5.3 Análisis Profundo de la Implementación

5.3.1 Dinámica de Composición

En la clase Factura, el atributo self._items es una lista que actúa como contenedor. La relación es de composición porque las instancias de LineaFactura se crean y almacenan con el propósito específico de servir a esa factura. No se espera que una LineaFactura sea compartida entre dos facturas diferentes. Esta estructura garantiza que la factura sea una unidad atómica y coherente. Si decidiéramos implementar un método eliminar_factura, podríamos limpiar explícitamente esta lista, y el recolector de basura de Python se encargaría de liberar la memoria de las líneas (suponiendo que no existan referencias externas), demostrando la gestión del ciclo de vida vinculada.[2]

5.3.2 El Poder de la Colaboración e Inyección

La CalculadoraImpuestos se pasa al constructor de la Factura. Observe que la factura no hace self.calc = CalculadoraImpuestos(). Al recibir el objeto desde fuera (inyección), desacoplamos la lógica de facturación de la lógica fiscal.

- **Beneficio**: Si mañana la ley cambia y el impuesto es complejo (dependiendo de la categoría del producto), podemos crear una CalculadoraImpuestosAvanzada y pasarla a la factura sin cambiar una sola línea de código dentro de la clase Factura. Esto cumple con el principio Abierto/Cerrado (Open/Closed Principle) de SOLID.

5.3.3 Robustez mediante Propiedades

La clase Producto utiliza @property para salvaguardar el precio. Es crucial notar cómo el constructor __init__ asigna a self.precio (la propiedad pública) y no a self._precio (la variable privada).

- **Mecanismo**: Al hacer self.precio = precio en el __init__, Python intercepta la asignación y la redirige al método decorado con @precio.setter.
- **Resultado**: Es imposible crear un objeto Producto en un estado inválido (precio negativo). Esto eleva la calidad del software de "scripting básico" a "ingeniería robusta".[13]

5.3.4 La Ilusión de la Sobrecarga

El método agregar_item maneja la complejidad de ofrecer múltiples interfaces. Utiliza isinstance para bifurcar la lógica.

- Si recibe un Producto, lo usa directamente.

- Si recibe un str, infiere que es un nombre y busca los argumentos opcionales precio.
  Esta técnica, aunque simula la sobrecarga, debe usarse con moderación. Si la lógica se vuelve muy compleja, es preferible usar métodos con nombres explícitos como agregar_producto_existente y crear_y_agregar_producto para mantener la claridad. Sin embargo, para este problema de baja complejidad, proporciona una experiencia de usuario (Developer Experience) muy fluida y "mágica".20

6. Perspectivas Avanzadas e Implicaciones

6.1 Costos y Beneficios de la Composición en Python

Aunque la composición proporciona una flexibilidad superior a la herencia, introduce una capa de indirección. En el ejemplo de la `Factura`, para obtener el total, la factura debe iterar sobre sus ítems, y cada ítem debe consultar a su producto. En escenarios de muy alto rendimiento (como sistemas de trading en tiempo real o motores de videojuegos), este "puntero tras puntero" puede incurrir en fallos de caché de CPU (cache misses). Sin embargo, para aplicaciones empresariales generales, los beneficios de mantenibilidad superan por órdenes de magnitud a estos costos computacionales despreciables.

6.2 El Cambio de Paradigma "Pythonico"

Históricamente, los Patrones de Diseño (Gang of Four) se escribieron pensando en C++ y Smalltalk. Las características de Python a menudo hacen obsoletos los patrones tradicionales o simplifican drásticamente su implementación.

- **Singleton**: Puede ser reemplazado por un módulo de Python (que es un singleton natural).
- **Factory**: Puede ser reemplazado por un método de clase, como se discutió en la sobrecarga de constructores.
- **Strategy**: (Como en el caso de la calculadora de impuestos) puede ser tan simple como pasar una función lambda en lugar de una clase completa, gracias a que las funciones son ciudadanos de primera clase en Python.[21]

6.3 Efectos de Segundo Orden en el Testing

Al usar Colaboración para la `CalculadoraImpuestos`, inadvertidamente preparamos el sistema para pruebas unitarias rigurosas. Al probar la clase `Factura`, no necesitamos una base de datos fiscal real; podemos pasar un "Mock" que siempre devuelva 0 o un valor fijo. Si la `Factura` hubiera codificado rígidamente la lógica fiscal (creando la calculadora internamente mediante composición estricta), probar casos de borde sería mucho más difícil. Esto revela una regla de diseño sutil: **Componer datos (Líneas, Productos), pero Colaborar con lógica (Servicios, Calculadoras).**

7. Conclusión

La transición de scripts procedimentales a un Diseño Orientado a Objetos robusto en Python requiere dominar no solo la sintaxis, sino la semántica de las relaciones y la gestión del estado. Al priorizar la **Composición sobre la Herencia**, los desarrolladores crean sistemas modulares y resistentes al cambio. Al utilizar la **Colaboración**, los sistemas se vuelven flexibles y testeables. Y al aprovechar las **Propiedades** y la **Sobrecarga Simulada**, las clases se vuelven seguras e intuitivas de usar, evitando la deuda técnica asociada con diseños rígidos o inseguros.

El "Sistema de Facturación" desarrollado en este informe sirve como un microcosmos de estos principios. Aunque es un problema de "baja complejidad", las decisiones arquitectónicas tomadas—validación en mutadores, firmas de métodos flexibles e inyección de servicios—son idénticas a las que se encuentran en frameworks de Python de escala empresarial como Django, FastAPI o SQLAlchemy. El dominio de estos bloques de construcción fundamentales es el umbral que separa a un programador de scripts de un ingeniero de software en Python.