1. **Herencia**

En el código, las clases “Comprador”, “Organizador”, “EventoParrillada”, “EventoVIP”, “Venta”, y “GestorVentas” hacen uso de la herencia. Por ejemplo, “Comprador” y “Organizador” son subclases de “Persona”, lo que significa que heredan sus atributos y métodos.

```
3
4   class Persona:
5       def __init__(self, nombre, DNI):
6           self.nombre = nombre
7           self.DNI = DNI
8
9
10  class Comprador(Persona):
11      def __init__(self, nombre, DNI):
12          super().__init__(nombre, DNI)
13
14
15  class Organizador(Persona):
16      def __init__(self, nombre, DNI):
17          super().__init__(nombre, DNI)
18
```

Polimorfismo

En el código, el polimorfismo se manifiesta en el método “mostrar_detalle()” definido en la interfaz “Evento”. Tanto “EventoParrillada” como “EventoVIP” implementan este método de acuerdo con sus propias necesidades, pero ambos métodos se pueden llamar de la misma manera desde el contexto de un objeto “Evento”.

Además, en el método “agregar_venta()” de la clase “GestorVentas”, se acepta cualquier objeto “Evento” como parte de la venta. Esto permite que el método sea polimórfico, ya que puede funcionar con cualquier subclase de “Evento”, ya sea “EventoParrillada”, “EventoVIP”, u otras clases que puedan ser creadas en el futuro.

```
28  class EventoParrillada(Evento):
29      def __init__(self, nombre, fecha, lugar, costo, descripcion):
30          self.nombre = nombre
31          self.fecha = fecha
32          self.lugar = lugar
33          self.costo = costo
34          self.descripcion = descripcion
35
36      def mostrar_detalle(self):
37          return f"Evento de Parrillada: {self.nombre}, Fecha: {self.fecha}, Lugar: {self.lugar}, Costo: ${self.costo}, Descripción: {self.descripcion}"
38
39  class EventoVIP(Evento):
40      def __init__(self, nombre, fecha, lugar, costo, descripcion, beneficios):
41          self.nombre = nombre
42          self.fecha = fecha
43          self.lugar = lugar
44          self.costo = costo
45          self.descripcion = descripcion
46          self.beneficios = beneficios
47
48      def mostrar_detalle(self):
49          return f"Evento VIP: {self.nombre}, Fecha: {self.fecha}, Lugar: {self.lugar}, Costo: ${self.costo}, Descripción: {self.descripcion}, Beneficios: {self.beneficios}
```

El diseño del sistema se beneficia de la herencia y el polimorfismo al permitir que nuevas clases se agreguen y se integren fácilmente en la estructura existente sin modificar el código existente.

Por ejemplo, si se desea agregar un nuevo tipo de evento, simplemente se puede crear una nueva clase que herede de “Evento” y proporcione su propia implementación De “mostrar_detalle()”. No se necesitan cambios en otras partes del código.

2. Las excepciones personalizadas se implementaron para mejorar la robustez del sistema al proporcionar un manejo más específico y significativo de los errores que pueden ocurrir durante la ejecución del programa. Aquí hay un ejemplo de como estas excepciones manejan casos de error:

   ArchivoNoEncontradoError y ArchivoInvalidoError:
   Estas excepciones se utilizan al cargar o guardar datos desde o hacia un archivo para manejar casos en los que el archivo no se encuentra o no es válido.
   Por ejemplo, al cargar un archivo de ventas, si el archivo especificado no se encuentra, se lanza ArchivoNoEncontradoError, mientras que si el archivo no es un archivo JSON válido, se lanza ArchivoInvalidoError.

```python
def cargar_ventas(self, archivo):
    try:
        with open(archivo, 'r') as f:
            data = json.load(f)
            self.ventas = [Venta(Comprador(**venta['comprador']), Evento(**venta['evento'])) for venta in data]
    except FileNotFoundError:
        raise ArchivoNoEncontradoError("El archivo especificado no se encontró.")
    except json.JSONDecodeError:
        raise ArchivoInvalidoError("El archivo especificado no es un archivo JSON válido.")
```

3. El proceso de serializar y deserializar objetos complejos con herencia a JSON implica convertir los objetos y sus atributos en una representación JSON y luego reconstruir los objetos a partir de esa representación JSON. Aquí está el proceso general y algunos desafíos asociados:

   Proceso de Serialización:
   Definir un formato JSON adecuado que represente todos los atributos relevantes de los objetos y sus relaciones de herencia.
   Recorrer la estructura de objetos y convertir cada atributo en su equivalente JSON.
   Utilizar bibliotecas de serialización JSON, como json en Python, para realizar la conversión de objetos a JSON.

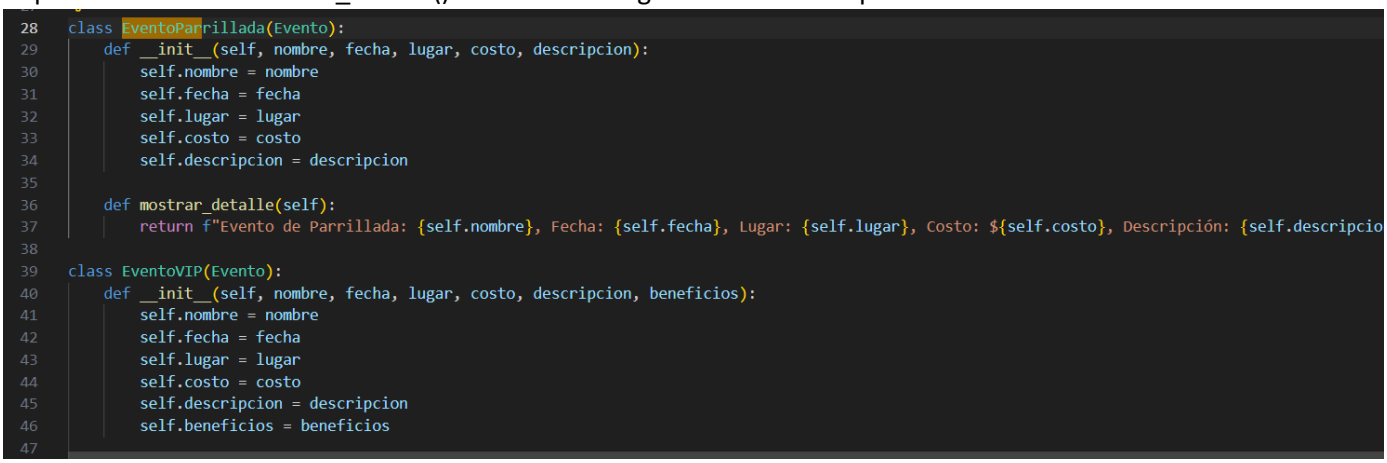
   Desafíos:
   El principal desafío radica en garantizar que se serialicen y deserialicen correctamente los atributos heredados de las clases base y subclases. Tuve que ver varios videos de youtube para entender.

   Y para asegurar la consistencia y completitud Se pueden utilizar pruebas exhaustivas para verificar que los objetos se serialicen y deserialicen correctamente en una variedad de escenarios y condiciones y la validación de los datos serializados al deserializarlos puede ayudar a detectar cualquier pérdida de información o inconsistencia.

```python
        total = sum(venta.calcular_total() for venta in self.ventas)
        return f"Total de ventas: ${total}"

    def guardar_ventas(self, archivo):
        with open(archivo, 'w') as f:
            json.dump([venta.__dict__ for venta in self.ventas], f)

    def cargar_ventas(self, archivo):
        try:
            with open(archivo, 'r') as f:
                data = json.load(f)
                self.ventas = [Venta(Comprador(**venta['comprador']), Evento(**venta['evento'])) for venta in data]
        except FileNotFoundError:
            raise ArchivoNoEncontradoError("El archivo especificado no se encontró.")
        except json.JSONDecodeError:
            raise ArchivoInvalidoError("El archivo especificado no es un archivo JSON válido.")

```

4. La estructura de clases proporcionada en el código facilita la adición de nuevos tipos de eventos y cambios en la lógica de ventas por algunas cositas que hice las cuales son.

   Abstracción y herencia: La jerarquía de clases bien definida, con una interfaz común Evento y diferentes implementaciones como EventoParrillada y EventoVIP, facilita la adición de nuevos tipos de eventos. Al heredar de la clase base Evento, cualquier nueva clase de evento puede proporcionar su propia implementación de mostrar_detalle() sin afectar la lógica central de la aplicación.



```python
class EventoParrillada(Evento):
    def __init__(self, nombre, fecha, lugar, costo, descripcion):
        self.nombre = nombre
        self.fecha = fecha
        self.lugar = lugar
        self.costo = costo
        self.descripcion = descripcion

    def mostrar_detalle(self):
        return f"Evento de Parrillada: {self.nombre}, Fecha: {self.fecha}, Lugar: {self.lugar}, Costo: ${self.costo}, Descripción: {self.descripcio

class EventoVIP(Evento):
    def __init__(self, nombre, fecha, lugar, costo, descripcion, beneficios):
        self.nombre = nombre
        self.fecha = fecha
        self.lugar = lugar
        self.costo = costo
        self.descripcion = descripcion
        self.beneficios = beneficios

```

   Flexibilidad en la creación de eventos: La implementación de métodos en GestorVentas para crear nuevos eventos, como crear_evento_parrillada() y crear_evento_vip(), facilita la adición de nuevos tipos de eventos. Estos métodos encapsulan la lógica de creación de eventos, lo que permite agregar nuevos tipos de eventos con mayor facilidad.

   Manejo de ventas y eventos por separado: Al separar la gestión de ventas en una clase dedicada GestorVentas, se permite una mayor flexibilidad para realizar cambios en la lógica de ventas sin afectar directamente a las clases de eventos. Esto facilita la modificación o extensión de la lógica de ventas sin impactar en la implementación de los eventos.

5. Apoyan de muchas maneras, la herencia, el polimorfismo ayudan muchísimo a mejorar tanto visualmente como mecánicamente el código, y la verdad pondría más pero son las 11:58 y ya no llego