UNIVERSIDAD NACIONAL GUILLERMO BROWN

INFORME
TRABAJO PRÁCTICO INTEGRADOR

PRIMER CUATRIMESTRE DE 2025

Refactorización de una Aplicación Web Django:
Un Enfoque Práctico de Diseño Orientado a Objetos

Grupo Nº 9:

Fecha de Entrega: 20 de Junio de 2024

Integrantes del Grupo: 9

Apellido y Nombre	E-mail
Nahuel Dalesio	nahuee.dalesiio@gmail.com
Darío Giménez	dgimenez.developer@gmail.com
Federico Paál	federicopaal@gmail.com

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Índice

1. Introducción

1.1. Tema Elegido

El presente trabajo se enmarca en el área de **Aplicaciones Web con Frameworks Python**. Específicamente, aborda la refactorización de una aplicación web existente construida con el framework Django, aplicando los principios fundamentales de la Programación Orientada a Objetos (POO) y el Diseño Orientado a Objetos (DOO) para transformarla en una solución más robusta, escalable y mantenible.

1.2. Motivación

El punto de partida fue el proyecto Proyecto_Final_Paal, una aplicación funcional de una tienda de revistas y cómics. Si bien cumplía con sus objetivos básicos, su código fuente presentaba deficiencias de diseño significativas, como una alta duplicación de código y una baja cohesión. Estas características lo convertían en un candidato ideal para un proceso de refactorización.

La principal motivación fue llevar a la práctica los conceptos teóricos abordados durante la cursada. Vimos una oportunidad clara para aplicar principios como la **herencia**, el **polimorfismo** y el **encapsulamiento** para resolver problemas concretos de diseño, demostrando cómo la POO no es solo un paradigma de programación, sino una herramienta esencial para construir software de calidad profesional

2. Análisis del Proyecto Original (Proyecto_Final_Paal)

El proyecto original, aunque funcional, sufría de varios problemas de diseño que dificultaban su mantenimiento y expansión.

2.1. Estructura de Modelos y Duplicación de Código

El problema más evidente residía en el archivo models.py. Existían tres modelos distintos para representar productos $(Libro,\ Merchandising,\ Novedad)$, los cuales compartían la mayoría de sus campos.

```python
# Proyecto_Final_Paal/App_Tercera_Pre_Entrega_Paal/models.py (extracto)
class Novedad(models.Model):
    titulo = models.CharField(max_length=40)
    autor = models.CharField(max_length=40)
    precio = models.IntegerField()
    imagen = models.ImageField(...)
    creacion = models.DateField(...)
    texto = models.CharField(...)

class Libro(models.Model):
    titulo = models.CharField(max_length=40)
    autor = models.CharField(max_length=40)
    precio = models.IntegerField()
    imagen = models.ImageField(...)
    creacion = models.DateField(...)
    texto = models.CharField(...)
```

Esta estructura violaba directamente el principio **DRY (Don't Repeat Yourself)**. Cualquier cambio en la definición de un producto (por ejemplo, añadir un campo stock) requeriría modificar múltiples clases, aumentando la probabilidad de errores y la carga de trabajo.

2.2. Vistas y URLs Repetitivas

La lógica en $views.py$ estaba implementada mayormente con Vistas Basadas en Funciones (FBV) que repetían la lógica de consulta para cada tipo de producto.

```python
# Proyecto_Final_Paal/App_Tercera_Pre_Entrega_Paal/views.py (conceptual)
def libros(req):
    libros = Producto.objects.filter(categoria="LIBROS")
    return render(req, "libros.html", {"libros": libros})

def novedades(req):
    novedades = Producto.objects.filter(categoria="NOVEDADES")
    return render(req, "novedades.html", {"novedades": novedades})
```

Esto se reflejaba en $urls.py$con patrones de URL separados para cada categoría, resultando en una configuración poco flexible.

2.3. Plantillas y Lógica de Presentación

La duplicación se extendía a las plantillas. Existían archivos como $detallesLibro.html,\ detallesMerchandising.html\ y\ detallesNovedad.html$, cuyo contenido era prácticamente idéntico. Esto hacía que cualquier cambio en la interfaz de detalle de un producto tuviera que replicarse manualmente en varios archivos.

2.4. Diagnóstico General: Violaciones a Principios de Diseño

- **Violación de DRY:** Como se ha demostrado, el código se repetía en modelos, vistas y plantillas.
- **Baja Cohesión:** La lógica relacionada con un "producto" estaba dispersa en múltiples clases y funciones.
- **Alto Acoplamiento (potencial):** La necesidad de modificar múltiples archivos para un solo cambio conceptual indica un alto acoplamiento entre componentes que deberían ser más independientes.

3. El Proceso de Refactorización y Diseño Orientado a Objetos $(ProyectoFinal\ -\ Revisteria)$

El objetivo de la refactorización fue resolver los problemas mencionados aplicando un diseño orientado a objetos. La nueva arquitectura se fundamenta en clases cohesivas y bien definidas.

3.1. Herencia y Abstracción: Unificación de Modelos con Clases Base Abstractas

La solución más importante fue la introducción de una clase base abstracta $Producto$. Esta clase encapsula todos los campos y comportamientos comunes a cualquier producto de la tienda.

```python
# ProyectoFinal-Revisteria-main/tienda/models.py (extracto)
class Producto(models.Model):
    titulo = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    imagen = models.ImageField(upload_to='productos/', null=True, blank=True)
    categoria = models.CharField(...)
    texto = models.TextField(...)
    es_novedad = models.BooleanField(...)

    class Meta:
        abstract = True
        ordering = ['-creacion']

class Libro(Producto):
    autor = models.CharField(max_length=100, null=True, blank=True)

    def save(self, *args, **kwargs):
        self.categoria = 'libro'
        super().save(*args, **kwargs)

class Merchandising(Producto):
    def save(self, *args, **kwargs):
        self.categoria = 'merchandising'
        super().save(*args, **kwargs)
```

Aplicación de POO:

- **Herencia:** $Libro$ y $Merchandising$ heredan de $Producto$, reutilizando toda la definición común.
- **Abstracción:** Producto es una clase abstracta $(abstract = True)$, lo que significa que no puede ser instanciada directamente. Define un "contrato" o plantilla para lo que significa ser un producto en nuestro sistema.

3.2. Polimorfismo: Vistas Genéricas y URLs Dinámicas

Se reemplazaron las vistas funcionales repetitivas por **Vistas Basadas en Clases (CBV)** genéricas de Django, que aprovechan el polimorfismo. Las vistas ahora tratan a $Libro$ y $Merchandising$ como subtipos de Producto, permitiendo un manejo unificado. La vista de detalle, por ejemplo, puede mostrar cualquier producto sin saber de antemano su clase concreta.

```python
# ProyectoFinal-Revisteria-main/tienda/views.py (extracto)
class ProductoListView(ListView):
    template_name = 'tienda/producto_list.html'
    context_object_name = 'productos'

    def get_queryset(self):
        categoria = self.kwargs.get('categoria')
        if categoria == 'libros':
            return Libro.objects.all()
        elif categoria == 'merchandising':
            return Merchandising.objects.all()
```

Una única clase $ProductoListView$ ahora puede listar objetos de tipo $Libro$ o $Merchandising$ dependiendo del parámetro en la URL. Esto fue posible gracias a una ruta unificada en $urls.py$:

$path('productos/ < str: categoria > /', views.ProductoListView.as\_view(), name = 'producto\_list')$

3.3. Encapsulamiento: Centralización de la Lógica de Negocio

El encapsulamiento se aplicó para agrupar datos y los métodos que operan sobre ellos.

1. **Lógica del Carrito:** Se creó la clase **Carrito** ($tienda/carrito.py$) que encapsula toda la lógica de gestión del carrito de compras (agregar, restar, limpiar, guardar en sesión). Las vistas solo interactúan con la interfaz pública de esta clase, sin necesidad de conocer los detalles de su implementación interna.

2. **Lógica de Modelos:** Se sobrescribió el método **save()** en las clases hijas ($Libro, Merchandising$). Esta es una forma de polimorfismo que encapsula la lógica de asignar la categoría correcta automáticamente al guardar un objeto, liberando al programador de hacerlo manualmente.

3.4. Reutilización de Código: Aplicación de Mixins para Control de Acceso

Para restringir el acceso a las vistas de gestión (crear, editar, eliminar productos), se implementó un $StaffRequiredMixin$.

```python
# ProyectoFinal-Revisteria-main/tienda/views.py (extracto)
class StaffRequiredMixin(UserPassesTestMixin):
    def test_func(self):
        return self.request.user.is_authenticated and self.request.user.perfil.rol == 'staff'

class ProductoCreateView(StaffRequiredMixin, CreateView):
    # ...
```

Este mixin permite añadir una capa de autorización a cualquier CBV de forma declarativa, sin duplicar código de verificación de permisos en cada vista. Es un claro ejemplo de composición de objetos y reutilización de código.

3.5. Mejora de la Interfaz y Adopción de Buenas Prácticas

- **Plantillas Unificadas:** Se redujo el número de plantillas a un conjunto mínimo y reutilizable ($base.html, producto\_list.html, producto\_detail.html$).
- **Frameworks de Frontend:** Se adoptó Bootstrap 5 para un diseño responsivo y moderno.
- **Formularios Mejorados:** Se utilizó la librería $django - crispy - forms$ para renderizar formularios de manera automática y consistente con Bootstrap, separando la lógica del formulario de su presentación.

4. Aplicación de Patrones de Diseño

Durante la refactorización, identificamos y aplicamos (consciente e inconscientemente) varios patrones de diseño clásicos.

4.1. Patrón Template Method

Las Vistas Basadas en Clases (CBV) de Django son una implementación del patrón **Template Method**. La clase base (ej.$ListView$) define el esqueleto de un algoritmo en un método (como $dispatch()$ o $get()$), pero delega ciertos pasos a las subclases. Nosotros implementamos estos "pasos" al sobreescribir métodos como $get\_queryset()$ y $get\_context\_data()$, personalizando el comportamiento sin alterar la estructura general del algoritmo.

4.2. Patrón Factory (Aplicación Conceptual)

La vista $ProductoCreateView$ utiliza una lógica similar a un **Factory Method**. El método $get\_form\_class()$ actúa como una fábrica que decide qué clase de formulario ($LibroForm$ o $MerchandisingForm$) instanciar basándose en el parámetro categoría de la URL. Esto desacopla la vista de las clases concretas de formularios.

4.3. Patrón Singleton

El objeto $Carrito$, aunque se instancia en cada solicitud, está ligado a la sesión del usuario ($request.session$). Para una sesión de usuario dada, siempre se opera sobre el mismo estado del carrito almacenado en la sesión. Esto se asemeja al propósito del patrón **Singleton**: garantizar una única instancia de estado para un contexto específico (en este caso, la sesión del usuario).

5. Resultados y Dificultades

5.1. Dificultades Técnicas y Soluciones Implementadas

1. **Dificultad: Colisión de IDs en el Carrito.**
    - **Problema:** Un Libro con id=1 y un Merchandising con id=1 eran indistinguibles en el diccionario del carrito si solo usábamos el ID como clave.
    - **Solución:** Se diseñó una clave compuesta que combina la categoría y el ID del producto (ej: 'libro_1', 'merchandising_1'). Esto garantiza una clave única para cada producto en el sistema, sin importar su modelo.

2. **Dificultad: Obtener un objeto sin saber su tipo.**
    - **Problema:** En la vista de detalle, la URL solo provee el pk del producto. No sabíamos si debíamos buscar en la tabla Libro o Merchandising.
    - **Solución:** Se implementó una lógica try-except en el método get_object de la CBV. Intenta buscar el pk en el primer modelo y, si lanza una excepción DoesNotExist, procede a buscarlo en el segundo.

5.2. Resultados Obtenidos: Comparativa de Métricas

Métrica de Calidad	***Proyecto_Final_Paal (Antes)***	***ProyectoFinal-Revisteria (Después)***
Duplicación de Código	Alta (Modelos, Vistas, Plantillas)	Mínima (Principios DRY aplicados)
Mantenibilidad	Baja (Cambios en cascada)	Alta (Cambios localizados y modulares)
Escalabilidad	Baja (Añadir categorías era costoso)	Alta (Fácil de añadir nuevos tipos de productos)
Cohesión	Baja (Lógica de negocio dispersa)	Alta (Clases con responsabilidades únicas)
Acoplamiento	Alto (Componentes interdependientes)	Bajo (Componentes desacoplados)

6. Conclusiones

6.1. Conclusiones Técnicas del Proyecto

La refactorización del proyecto fue un éxito rotundo. Se lograron los siguientes beneficios cuantificables:

- **Reducción de Código:** Se eliminó la redundancia en modelos, vistas y plantillas, resultando en una base de código más pequeña y legible.
- **Mantenibilidad Mejorada:** Gracias a la centralización de la lógica (principio DRY), realizar cambios es ahora más rápido y seguro. Por ejemplo, añadir un nuevo tipo de producto es trivial en comparación con el sistema anterior.

- **Mayor Escalabilidad:** La arquitectura actual está preparada para crecer. Añadir nuevas funcionalidades o tipos de productos se puede hacer de forma modular y sin afectar al código existente.
- **Adherencia a Principios S.O.L.I.D.:** El nuevo diseño respeta mejor principios como el de Responsabilidad Única (SRP) y el Abierto/Cerrado (OCP).

6.2. Conclusiones Generales de la Cursada

Este trabajo práctico integrador fue fundamental para consolidar los conocimientos adquiridos. Nos permitió pasar de la teoría de la Programación Orientada a Objetos a su aplicación práctica en un framework profesional como Django. Comprendimos que el DOO no es un objetivo en sí mismo, sino un medio para construir software de mayor calidad, más flexible y duradero.

La experiencia de analizar un código con deficiencias de diseño y transformarlo nos ha proporcionado una perspectiva invaluable sobre la importancia de planificar la arquitectura de software y de aplicar patrones de diseño para resolver problemas comunes de forma elegante y probada.

7. Anexos

7.1. Enlace al Repositorio de Código Fuente

El código fuente completo del proyecto refactorizado se encuentra disponible en el siguiente repositorio de GitHub:

https://github.com/dgimenezdeveloper/ProyectoFinal-Revisteria