UNIVERSIDAD TECNOLÓGICA DE CHIHUAHUA

Desarrollo y Gestión de Software.

Fundamentos de arquitecturas de software

JUSTIFICACIÓN DE PATRONES DE DISEÑO

PRESENTA:

Angel David Arras Orozco

1123150014

DOCENTE:

Enrique Mascote

1. INTRODUCCIÓN

1.1 Descripción del Proyecto

El presente proyecto consiste en el desarrollo de un **Sistema de Gestión de Restaurante** implementado con el framework Django (Python), que permite administrar de manera integral las operaciones cotidianas de un establecimiento de alimentos y bebidas.

El sistema abarca desde la gestión del menú y mesas, hasta el procesamiento completo de pedidos, pagos y generación de reportes, todo ello implementando tres patrones de diseño fundamentales que mejoran la arquitectura, mantenibilidad y escalabilidad del software.

1.2 Objetivo del Prototipo

Demostrar la aplicación práctica de patrones de diseño en un sistema real, implementando al menos un patrón de cada categoría (creacional, estructural y de comportamiento) que resuelvan problemas específicos de arquitectura de software y aporten valor tangible al sistema.

1.3 Alcance

El sistema desarrollado incluye las siguientes funcionalidades principales:

- **Gestión de Categorías y Platillos:** CRUD completo con imágenes, precios y tiempos de preparación
- **Gestión de Mesas:** Control de disponibilidad y capacidad
- **Gestión de Pedidos:** Ciclo completo desde creación hasta pago
- **Sistema de Notificaciones:** Alertas automáticas a cocina, meseros y administradores
- **Configuración Centralizada:** Parámetros globales del restaurante (impuestos, propinas, horarios)
- **Control de Acceso por Roles:** Permisos diferenciados para administradores, meseros, cocina y caja
- **Cálculos Automáticos:** Subtotales, impuestos y propinas

2. PATRÓN SINGLETON - Gestor de Configuración

2.1 Categoría

Creacional

2.2 Problema que Resuelve

El restaurante necesita mantener una configuración única y consistente en todo el sistema. Parámetros como el porcentaje de impuesto (IVA), propina sugerida, horarios de operación y capacidad máxima deben ser los mismos independientemente de qué módulo o vista los consulte.

Problemas sin el patrón:

- Múltiples instancias de configuración con valores diferentes
- Inconsistencias en cálculos de totales e impuestos
- Desperdicio de memoria creando objetos de configuración repetidamente
- Dificultad para actualizar configuración de forma centralizada

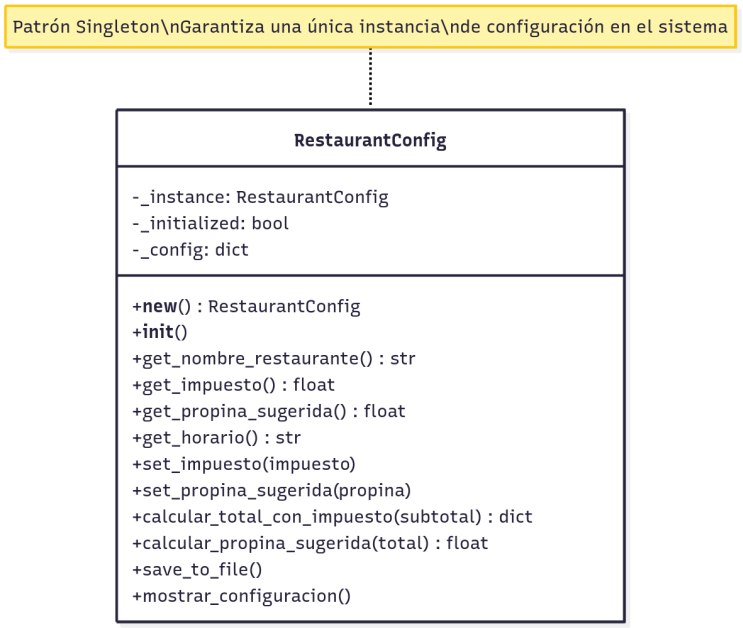
2.3 Implementación

Ubicación: `restaurant/patterns/singleton/config_manager.py`

Código clave:

```
class RestaurantConfig:
    _instance = None
    _initialized = False

    def __new__(cls):
        if cls._instance is None:
            print("🔧 Creando NUEVA instancia de RestaurantConfig...")
            cls._instance = super(RestaurantConfig, cls).__new__(cls)
        else:
            print("♻️ Reutilizando instancia existente")
        return cls._instance

    def __init__(self):
        if not RestaurantConfig._initialized:
            self._config = {
                'nombre_restaurante': 'Restaurante Code & Taste',
                'impuesto': 0.16,
                'propina_sugerida': 0.15,
                'horario': '9:00 AM - 11:00 PM',
                'capacidad_maxima': 80
            }
            RestaurantConfig._initialized = True
```

2.4 Diagrama UML



2.5 Uso en el Sistema

Ejemplo 1 - Cálculo de totales en pedidos:

```
# En models.py - Pedido
def calcular_totales(self):
    config = RestaurantConfig()  # Obtiene la instancia única
    self.subtotal = sum(item.subtotal for item in self.items.all())
    self.impuesto = self.subtotal * config.get_impuesto()
    self.total = self.subtotal + self.impuesto
    self.save()
```

Ejemplo 2 - Vista de configuración:

```
# En views.py
def ver_configuracion(request):
    config = RestaurantConfig()  # Misma instancia
    if request.method == 'POST':
        config.set_impuesto(float(request.POST.get('impuesto')) / 100)
        config.save_to_file()  # Persiste cambios
```

2.6 Ventajas de Usar Singleton en este Contexto

1. **Consistencia Global:** Todas las vistas y modelos usan la misma configuración
2. **Punto Único de Actualización:** Cambiar el impuesto actualiza todo el sistema instantáneamente
3. **Eficiencia de Memoria:** Solo existe una instancia en memoria durante toda la ejecución

4. **Persistencia Opcional:** La configuración puede guardarse en archivo JSON para mantenerla entre sesiones
5. **Facilidad de Pruebas:** Es sencillo verificar que la configuración es única

2.7 Alternativas Consideradas

Opción 1: Variables globales

- **Rechazada:** Difícil de mantener, no encapsula lógica, problemas con importaciones circulares

Opción 2: Base de datos con una sola fila

- **Rechazada:** Overhead innecesario de consultas SQL para cada acceso a configuración

Opción 3: Archivo de configuración JSON directo

- **Rechazada:** Requiere lectura de archivo en cada acceso, no garantiza unicidad

Singleton elegido: Combina lo mejor de todas: instancia única en memoria, métodos encapsulados y opción de persistencia.

3. PATRÓN DECORATOR - Decoradores de Vistas

3.1 Categoría

Estructural

3.2 Problema que Resuelve

Las vistas de Django necesitan funcionalidades transversales como:

- Registro de accesos (logging)
- Control de permisos por rol
- Medición de rendimiento
- Validación de horarios

Problemas sin el patrón:

- Código repetitivo en cada vista
- Violación del principio DRY (Don't Repeat Yourself)
- Dificultad para añadir nueva funcionalidad a múltiples vistas
- Mezcla de responsabilidades (lógica de negocio + cross-cutting concerns)

3.3 Implementación

Ubicación: `restaurant/patterns/decorator/view_decorators.py`

Decorador 1 - Log de Accesos:

```
def log_view_access(view_func):
    @wraps(view_func)
    def wrapper(request, *args, **kwargs):
        user = request.user.username if request.user.is_authenticated else 'Anónimo'
        view_name = view_func.__name__
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        log_message = f"[{timestamp}] 👤 Usuario: {user} | 🎯 Vista: {view_name}"
        logger.info(log_message)
        print(f"\n📝 [DECORATOR LOG] {log_message}\n")

        return view_func(request, *args, **kwargs)
    return wrapper
```

Decorador 2 - Control de Roles:

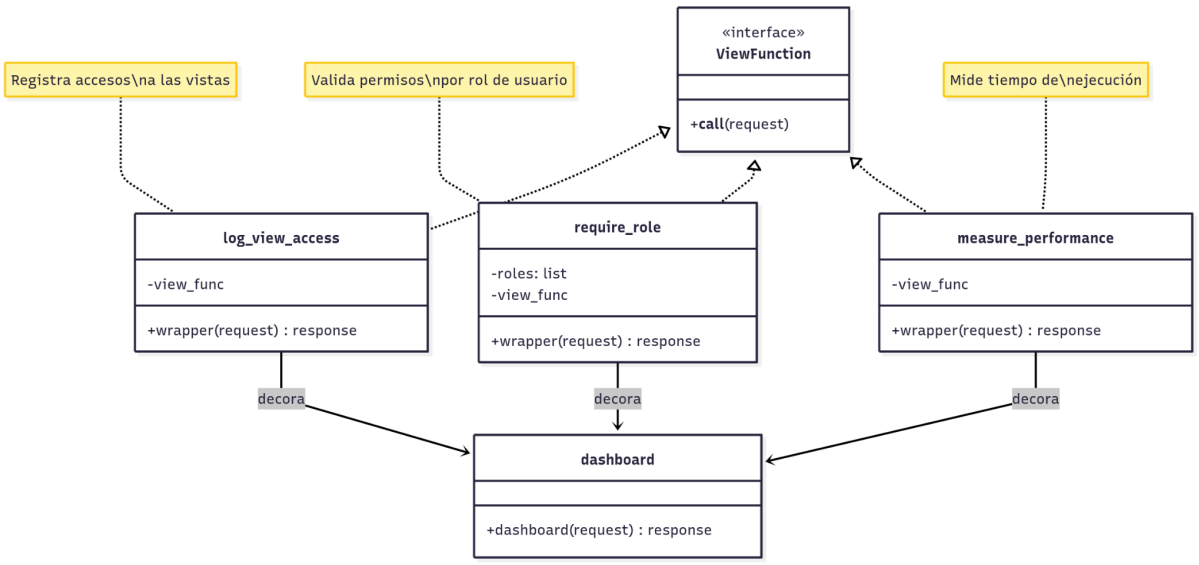
```
def require_role(*roles):
    def decorator(view_func):
        @wraps(view_func)
        @login_required
        def wrapper(request, *args, **kwargs):
```

```
        user_groups = request.user.groups.values_list('name', flat=True)
        has_permission = any(role in user_groups for role in roles)

        if not has_permission:
            messages.error(request, 'No tienes permisos para esta sección')
            return redirect('restaurant:dashboard')

        return view_func(request, *args, **kwargs)
    return wrapper
return decorator
```

Decorador 3 - Medición de Rendimiento:

```
def measure_performance(view_func):
    @wraps(view_func)
    def wrapper(request, *args, **kwargs):
        start_time = time.time()
        response = view_func(request, *args, **kwargs)
        execution_time = time.time() - start_time

        logger.info(f"⏱️ Vista: {view_func.__name__} | Tiempo: {execution_time:.4f}s")
        return response
    return wrapper
```

3.4 Diagrama UML



3.5 Uso en el Sistema

Ejemplo - Vista decorada con múltiples decoradores:

```
@login_required
@log_view_access              # Registra acceso
@require_role('mesero', 'admin')  # Solo meseros y admins
@measure_performance          # Mide tiempo de ejecución
def crear_pedido(request, mesa_id=None):
    # Lógica de la vista sin código adicional
    if request.method == 'POST':
        # ... crear pedido
    return render(request, 'restaurant/pedidos/form.html')
```

Salida en consola al acceder a la vista:
```
📝 [DECORATOR LOG] [2024-11-28 22:15:30] 👤 Usuario: mesero1 | 🎯 Vista: crear_pedido
✅ [DECORATOR] Acceso autorizado: mesero1 accedió con rol válido
🚀 [DECORATOR PERFORMANCE] RÁPIDO! ⏱️ Vista: crear_pedido | Tiempo: 0.0234s
```

3.6 Ventajas de Usar Decorator en este Contexto

1. **Separación de Responsabilidades:** Las vistas solo contienen lógica de negocio
2. **Reutilización de Código:** Los decoradores se aplican a múltiples vistas
3. **Cumple Open/Closed:** Se añade funcionalidad sin modificar código existente
4. **Composición Flexible:** Se pueden combinar decoradores según necesidades
5. **Facilita Auditoría:** El logging centralizado permite rastrear accesos
6. **Mejora Seguridad:** Control de acceso consistente en todo el sistema

3.7 Alternativas Consideradas

Opción 1: Middleware de Django

- Rechazada: Aplica a TODAS las vistas, no permite selectividad

Opción 2: Código repetido en cada vista

- Rechazada: Viola DRY, difícil de mantener

Opción 3: Herencia de clases base

- Rechazada: Django usa vistas basadas en funciones, herencia complica la estructura

Decorator elegido: Es el enfoque idiomático de Python/Django, flexible y mantenible.

4. PATRÓN OBSERVER - Sistema de Notificaciones

4.1 Categoría

Comportamiento

4.2 Problema que Resuelve

Cuando ocurren eventos importantes en un pedido (creación, cambio de estado, pago), diferentes partes del sistema necesitan ser notificadas:

- **Cocina** debe saber de nuevos pedidos y cancelaciones
- **Meseros** deben saber cuando un pedido está listo
- **Administrador** debe monitorear todos los eventos

Problemas sin el patrón:

- Acoplamiento fuerte entre Pedido y sistemas de notificación
- Código repetitivo para cada tipo de notificación
- Dificultad para añadir nuevos observadores
- Lógica de notificación mezclada con lógica de negocio

4.3 Implementación

Ubicación: `restaurant/patterns/observer/observers.py`

Interface Observer:

```
class Observer(ABC):
    @abstractmethod
    def actualizar(self, pedido, evento, datos=None):
        pass
```

Sujeto (Subject):

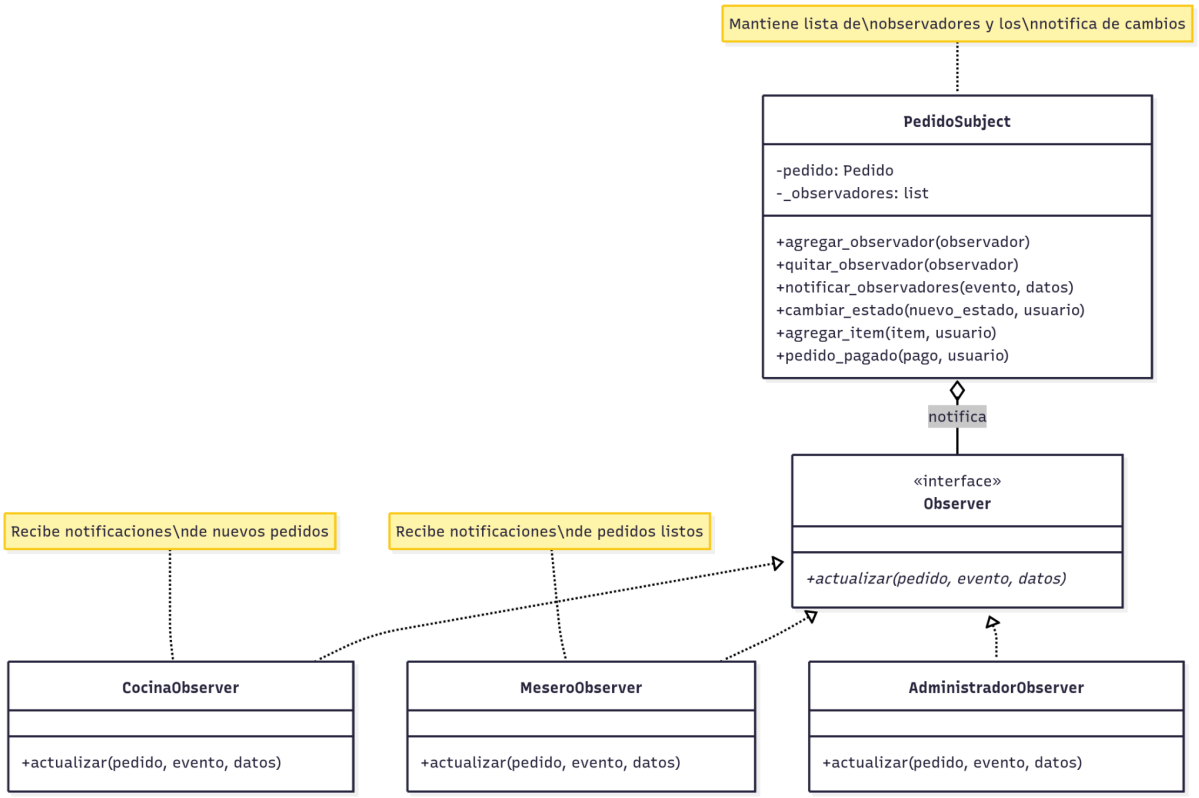
```
class PedidoSubject:
    def __init__(self, pedido):
        self.pedido = pedido
        self._observadores = []

    def agregar_observador(self, observador):
        if observador not in self._observadores:
            self._observadores.append(observador)

    def notificar_observadores(self, evento, datos=None):
        for observador in self._observadores:
            observador.actualizar(self.pedido, evento, datos)

    def cambiar_estado(self, nuevo_estado, usuario=None):
```

```
        estado_anterior = self.pedido.estado
        self.pedido.estado = nuevo_estado
        self.pedido.save()

        datos = {
            'estado_anterior': estado_anterior,
            'estado_nuevo': nuevo_estado,
            'usuario': usuario.username if usuario else 'Sistema'
        }

        self.notificar_observadores('cambio_estado', datos)
```

Observadores Concretos:

```
class CocinaObserver(Observer):
    def actualizar(self, pedido, evento, datos=None):
        if evento == 'cambio_estado' and datos['estado_nuevo'] == 'pendiente':
            print(f"🍳 [COCINA] NUEVO PEDIDO - Mesa {pedido.mesa.numero}")
            for item in pedido.items.all():
                print(f"   - {item.cantidad}x {item.platillo.nombre}")

class MeseroObserver(Observer):
    def actualizar(self, pedido, evento, datos=None):
        if evento == 'cambio_estado' and datos['estado_nuevo'] == 'listo':
            print(f"👨‍🍳 [MESERO] PEDIDO LISTO - Mesa {pedido.mesa.numero}")
```

4.4 Diagrama UML



4.5 Uso en el Sistema

Ejemplo - Crear pedido con notificaciones:

```
def crear_pedido(request, mesa_id=None):
    if request.method == 'POST':
        form = PedidoForm(request.POST)
        if form.is_valid():
            pedido = form.save(commit=False)
            pedido.mesero = request.user
            pedido.save()

            # PATRÓN OBSERVER
            pedido_subject = PedidoSubject(pedido)
            pedido_subject.agregar_observador(CocinaObserver())
            pedido_subject.agregar_observador(MeseroObserver())
            pedido_subject.agregar_observador(AdministradorObserver())

            # Al cambiar estado, todos son notificados automáticamente
            pedido_subject.cambiar_estado('pendiente', request.user)
```

Salida en consola:

```
➕ [OBSERVER] CocinaObserver suscrito al Pedido #1
➕ [OBSERVER] MeseroObserver suscrito al Pedido #1
➕ [OBSERVER] AdministradorObserver suscrito al Pedido #1

📣 NOTIFICACIÓN - Pedido #1 - Evento: cambio_estado

🍳 [COCINA] NUEVO PEDIDO - Mesa 3 - Pedido #1
  Items:
  - 1x Hamburguesa Clásica
  - 1x Refresco

👨‍🍳 [MESERO] ...
👔 [ADMIN] Pedido #1 - Estado: None → pendiente - Mesa 3
```

4.6 Ventajas de Usar Observer en este Contexto

1. **Desacoplamiento:** Pedido no conoce los detalles de los observadores
2. **Escalabilidad:** Fácil agregar nuevos observadores (EmailObserver, SMSObserver)
3. **Notificaciones Automáticas:** No se requiere código manual para cada notificación
4. **Flexibilidad:** Cada observador decide qué eventos le interesan
5. **Mantenibilidad:** Cambios en un observador no afectan a otros
6. **Trazabilidad:** Todas las notificaciones quedan registradas en logs

4.7 Alternativas Consideradas

Opción 1: Llamadas directas desde Pedido

```
# ❌ Rechazada
def cambiar_estado(self, nuevo_estado):
    self.estado = nuevo_estado
    notificar_cocina(self)  # Acoplamiento fuerte
    notificar_mesero(self)  # Difícil de mantener
```

Opción 2: Señales de Django (signals)

- Considerada pero no ideal: Menos explícito, debugging complicado

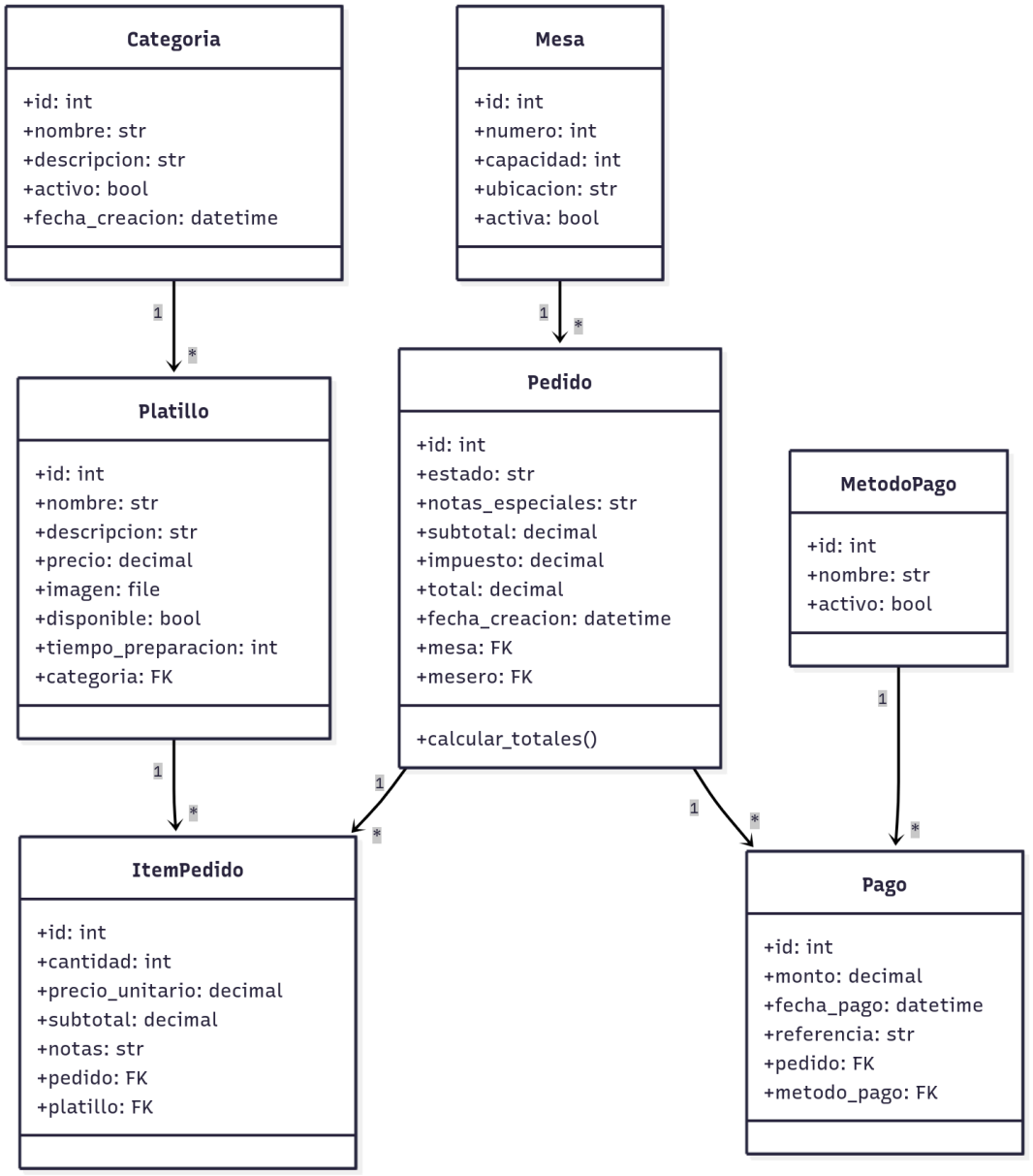
Opción 3: Sistema de colas (Celery)

- Rechazada: Complejidad excesiva para este alcance

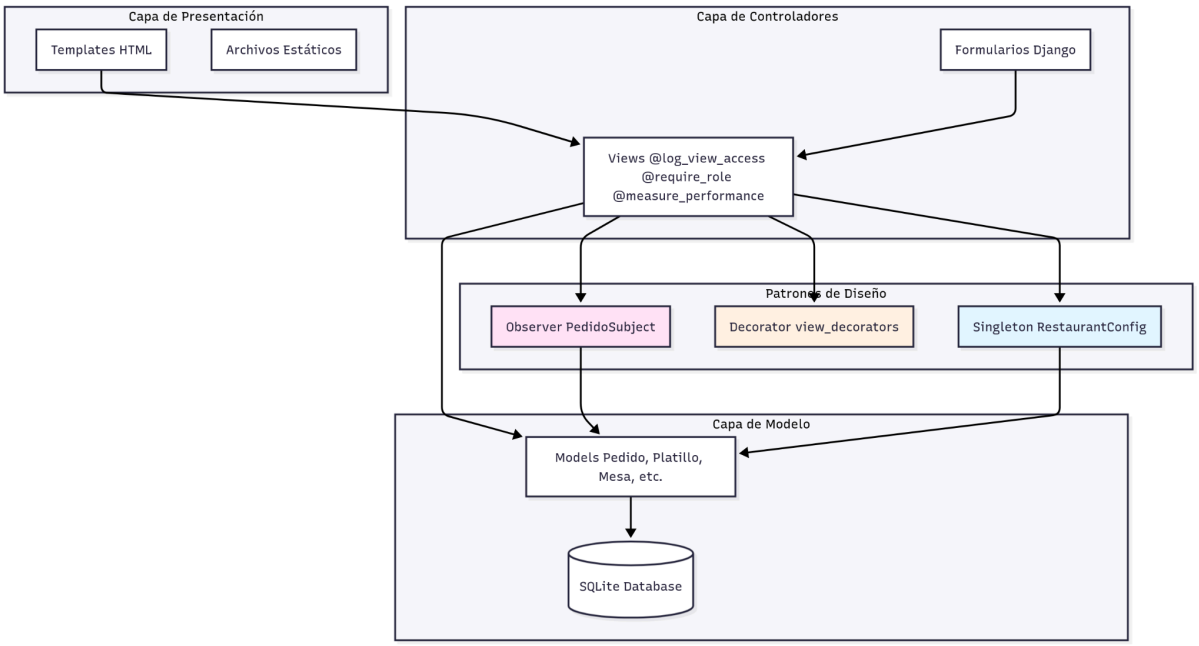
Observer elegido: Balance perfecto entre simplicidad y funcionalidad, patrón explícito y educativo.

5. ARQUITECTURA GENERAL DEL SISTEMA

5.1 Diagrama de Clases Completo



5.2 Diagrama de Componentes



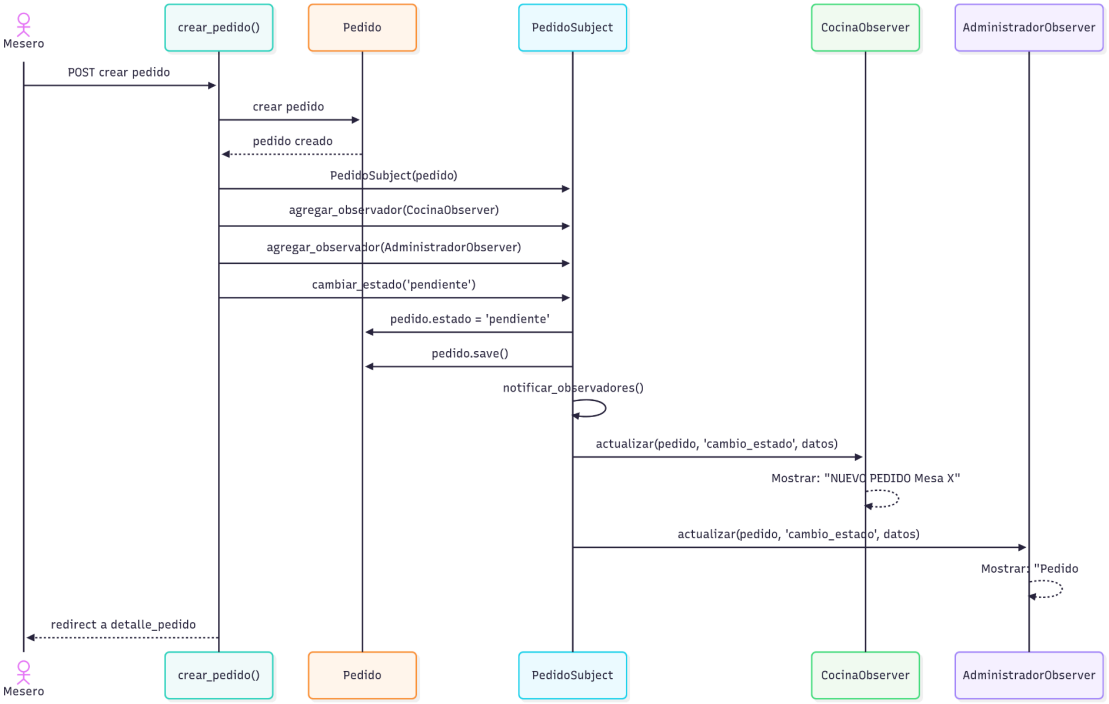
5.3 Interacción entre Patrones

Los tres patrones trabajan de forma complementaria:

Flujo típico al crear un pedido:

1. **Decorator** valida permisos y registra acceso a la vista
2. **Singleton** proporciona configuración para calcular impuestos
3. **Observer** notifica a cocina y meseros del nuevo pedido

Diagrama de secuencia:


Mesero
crear_pedido()
Pedido
PedidoSubject
CocinaObserver
AdministradorObserver
POST crear pedido
crear pedido
pedido creado
PedidoSubject(pedido)
agregar_observador(CocinaObserver)
agregar_observador(AdministradorObserver)
cambiar_estado('pendiente')
pedido.estado = 'pendiente'
pedido.save()
notificar_observadores()
actualizar(pedido, 'cambio_estado', datos)
Mostrar: "NUEVO PEDIDO Mesa X"
actualizar(pedido, 'cambio_estado', datos)
Mostrar: "Pedido
redirect a detalle_pedido
Mesero
crear_pedido()
Pedido
PedidoSubject
CocinaObserver
AdministradorObserver

6. CONCLUSIONES

6.1 Aprendizajes Obtenidos

1. **Los patrones resuelven problemas reales:** No son solo teoría académica, cada patrón implementado resolvió un problema específico del sistema.
2. **Los patrones se complementan:** Singleton + Decorator + Observer trabajando juntos crean una arquitectura más robusta que cada uno por separado.
3. **Mejoran la mantenibilidad:** Cuando fue necesario agregar logging a todas las vistas, solo se requirió crear un decorador y aplicarlo, en lugar de modificar 20+ vistas.
4. **Facilitan el testing:** Cada patrón encapsula su lógica, permitiendo pruebas unitarias independientes.
5. **Documentación viva:** El código con patrones bien implementados es autodocumentado, otros desarrolladores pueden entender la intención.

6.2 Dificultades Encontradas y Soluciones

Dificultad 1: Singleton en Python

- **Problema:** Python no tiene constructores privados como Java
- **Solución:** Uso del método `__new__()` para controlar la creación de instancias

Dificultad 2: Decoradores anidados

- **Problema:** Orden de aplicación afecta el comportamiento
- **Solución:** Documentar el orden correcto: `@login_required` → `@log_view_access` → `@require_role`

Dificultad 3: Observer y ORM de Django

- **Problema:** Notificaciones duplicadas en operaciones de actualización
- **Solución:** Flags de control y verificación de estado anterior vs nuevo

Dificultad 4: Persistencia de Singleton

- **Problema:** Configuración se perdía al reiniciar servidor
- **Solución:** Implementación de `save_to_file()` y carga automática en `__init__()`

6.3 Posibles Mejoras Futuras

1. **Patrón Strategy:** Para diferentes métodos de cálculo de descuentos
2. **Patrón Factory:** Para crear diferentes tipos de reportes
3. **Patrón Command:** Para implementar deshacer/rehacer en pedidos
4. **Patrón Chain of Responsibility:** Para validaciones complejas de pedidos
5. **Integración con Celery:** Para notificaciones asíncronas por email/SMS
6. **Testing automatizado:** Pruebas unitarias para cada patrón

7. **API RESTful:** Usando Django REST Framework
8. **Dashboard en tiempo real:** Con WebSockets para notificaciones instantáneas

6.4 Reflexión Final

La implementación de estos tres patrones de diseño transformó lo que podría haber sido un sistema monolítico difícil de mantener, en una aplicación modular, escalable y bien estructurada. Cada patrón aportó valor específico:

- **Singleton** garantizó consistencia en la configuración
- **Decorator** eliminó código repetitivo y separó responsabilidades
- **Observer** desacopló componentes y facilitó la extensibilidad

Este proyecto demuestra que los patrones de diseño no son solo conceptos teóricos, sino herramientas prácticas que mejoran significativamente la calidad del software cuando se aplican correctamente.