**5.7.1 Aspectos clave del diseño**

El diseño del sistema se basa en una arquitectura modular y flexible, fundamentada en la aplicación de patrones de diseño ampliamente aceptados en la ingeniería de software. Esto permite no solo una implementación coherente y bien estructurada, sino también una solución escalable y mantenible a largo plazo.

Uno de los aspectos clave del diseño es la adopción del patrón **Modelo-Vista-Controlador (MVC)**. Este patrón se refleja directamente en la organización del sistema: por ejemplo, las clases como cliente1, platillo1, carrito1, pedido1 y inventario1 representan el **modelo**, ya que encapsulan la lógica de negocio y el estado de los datos. La **vista**, en un entorno gráfico o web, sería responsable de mostrar la información almacenada en estas clases al usuario (como los platillos disponibles o los pedidos en curso). El **controlador**, que estaría representado por clases como PedidoController o CarritoController (en una implementación orientada), se encargaría de procesar las acciones del usuario (agregar platillos, confirmar pedidos, etc.) y coordinar los modelos y las vistas.

Para reforzar la comunicación entre componentes, se emplea el patrón **Observer**. Este patrón es esencial para actualizar automáticamente a los usuarios del sistema sobre eventos importantes. Por ejemplo, cuando un pedido (pedido1) cambia de estado a "Listo", tanto el cocinero (cocinero1) como el administrador (administrador1) son notificados. Esta dinámica es clave para mantener el flujo de trabajo ágil y actualizado sin necesidad de comprobaciones constantes.

Además, el patrón **Singleton** se aplica en clases que deben tener una única instancia global, como informe1, encargada de generar reportes del sistema. Solo debe existir una instancia de esta clase para evitar duplicidad en la generación de informes o inconsistencias en la lectura de datos administrativos, garantizando así una fuente única de verdad.

Otro patrón presente es la **Inyección de Dependencias**, especialmente útil al vincular componentes como el inventario (inventario1) con los platillos (platillo1, platillo2). En lugar de que las clases se instancien entre sí de forma directa, las dependencias se inyectan desde el exterior, permitiendo mayor flexibilidad, facilidad de pruebas y bajo acoplamiento. Por ejemplo, un servicio InventarioService podría inyectar dinámicamente los ingredientes (ingrediente1, ingrediente2) necesarios a los distintos platillos según la disponibilidad.

La arquitectura del sistema permite seguir la trazabilidad completa del pedido desde que el cliente (cliente1, como “Uriel Oswaldo”) agrega productos al carrito1, hasta que se confirma como pedido1, es preparado por cocinero1, revisado por administrador1, asociado a informe1 y finalmente se genera el ticket1 correspondiente. Esta secuencia lógica está respaldada por relaciones claras entre clases y el uso estratégico de patrones.

Por último, los principios SOLID están presentes en el diseño: por ejemplo, cada clase como cliente1, platillo1 o ticket1 cumple con el **principio de responsabilidad única**, ya que cada una tiene un propósito bien definido. Asimismo, el sistema está diseñado para ser extensible sin modificar las clases existentes, lo que refleja el **principio abierto/cerrado**. Todo esto contribuye a una estructura sólida, preparada para adaptarse a nuevas funcionalidades como métodos de pago, historial de pedidos, o reportes gráficos

 5.7.2 Elementos de diseño

Los elementos de diseño del sistema reflejan la estructura arquitectónica centrada en patrones de colaboración, clases especializadas, conectores abstractos y roles funcionales claramente definidos. A continuación, se describen los distintos componentes conforme a la norma:

 Entidades de diseño

- Colaboración: El sistema exhibe múltiples colaboraciones entre entidades clave. Por ejemplo, la colaboración entre las clases cliente1 y carrito1 permite capturar la lógica de selección de productos; a su vez, la clase pedido1 colabora con platillo1 e inventario1 para generar la solicitud de cocina.

- Clase: Las clases principales en el sistema incluyen cliente1, carrito1, pedido1, platillo1, ingrediente1, cocinero1, administrador1, informe1 y ticket1. Cada una representa una unidad funcional con responsabilidades específicas, alineadas al principio de responsabilidad única (SRP).

- Conector: Existen conectores lógicos que permiten la interacción entre entidades. Por ejemplo, el conector entre carrito1 y pedido1 representa la transición de elementos seleccionados hacia un pedido formal. Otro conector importante es entre pedido1 e informe1, que establece la transferencia de estado para reportes administrativos.

- Rol: Cada entidad cumple un rol claro dentro de una colaboración. cliente1 actúa como iniciador de la interacción; cocinero1 como ejecutor del pedido; informe1 como agregador de datos. Los roles definen comportamientos, no estructuras, lo que permite extender funcionalidades sin alterar la arquitectura base.

- Plantilla de marco (framework): Se utiliza el marco arquitectónico MVC, donde platillo1, carrito1 y pedido1 fungen como modelo; las vistas serían formularios y pantallas de gestión; y el controlador sería una capa lógica que orquesta las operaciones, posiblemente con clases como PedidoController o InventarioService. También se planea utilizar un framework de desarrollo como Spring Boot o Laravel para facilitar la implementación modular con soporte para patrones como inyección de dependencias y manejo de eventos.

- Patrón: Se aplican varios patrones:

MVC para separar presentación, lógica y datos.

Observer para notificar estados de pedidos a cocineros y administradores.

Singleton en informe1 para generar reportes únicos.

Inyección de dependencias entre servicios e inventarios.

Relaciones de diseño

Asociación: cliente1 está asociado a carrito1 mediante una relación directa uno-a-uno. pedido1 está asociado a uno o varios platillos (platillo1, platillo2), y también a un ticket1.

- Uso colaborativo: cocinero1 y administrador1 usan los datos de pedido1 para sus funciones. carrito1 colabora con cliente1 para construir una orden válida. informe1 colabora con pedido1 y ticket1 para consolidar estadísticas del sistema.

- Conector: Los conectores actúan como vínculos lógicos entre las entidades. Por ejemplo, el cambio de estado en pedido1 (de “en preparación” a “listo”) activa un conector de evento que comunica esta información a cocinero1 y administrador1. Este conector puede implementarse usando el patrón Observer.

Atributos de diseño

Nombre: Cada entidad está correctamente identificada en el modelo: cliente1, carrito1, platillo1, pedido1, etc. Esto facilita el rastreo de interacciones y mantiene el sistema coherente.

Restricciones de diseño

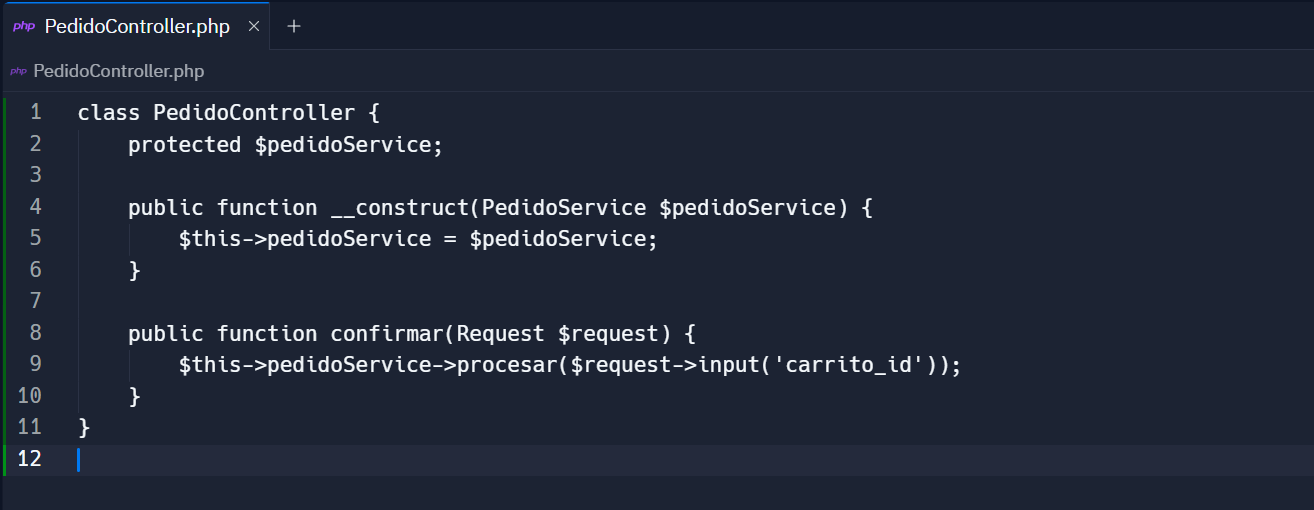
- Restricciones de colaboración: Algunas relaciones están restringidas. Por ejemplo:

* Un cliente1 solo puede tener un carrito1 activo.
* Un pedido1 no puede generarse si el carrito1 está vacío.
* informe1 solo puede generarse a partir de pedidos completados.
* Un cocinero1 no puede modificar el pedido una vez que se genera el ticket1.

Estas restricciones se implementarán mediante validaciones internas en los métodos del sistema y mediante reglas de negocio en la lógica de controladores.

Patrón: Se aplican varios patrones:  
  
**Modelo-Vista-Controlador** (MVC)  
El sistema sigue el patrón MVC para organizar la lógica. Las clases cliente1, carrito1, platillo1 y pedido1 representan el Modelo. Los formularios de entrada y salida (en interfaz web o móvil) representan las Vistas. La lógica de control se centraliza en controladores como PedidoController.

Ejemplo (controlador procesando un pedido):

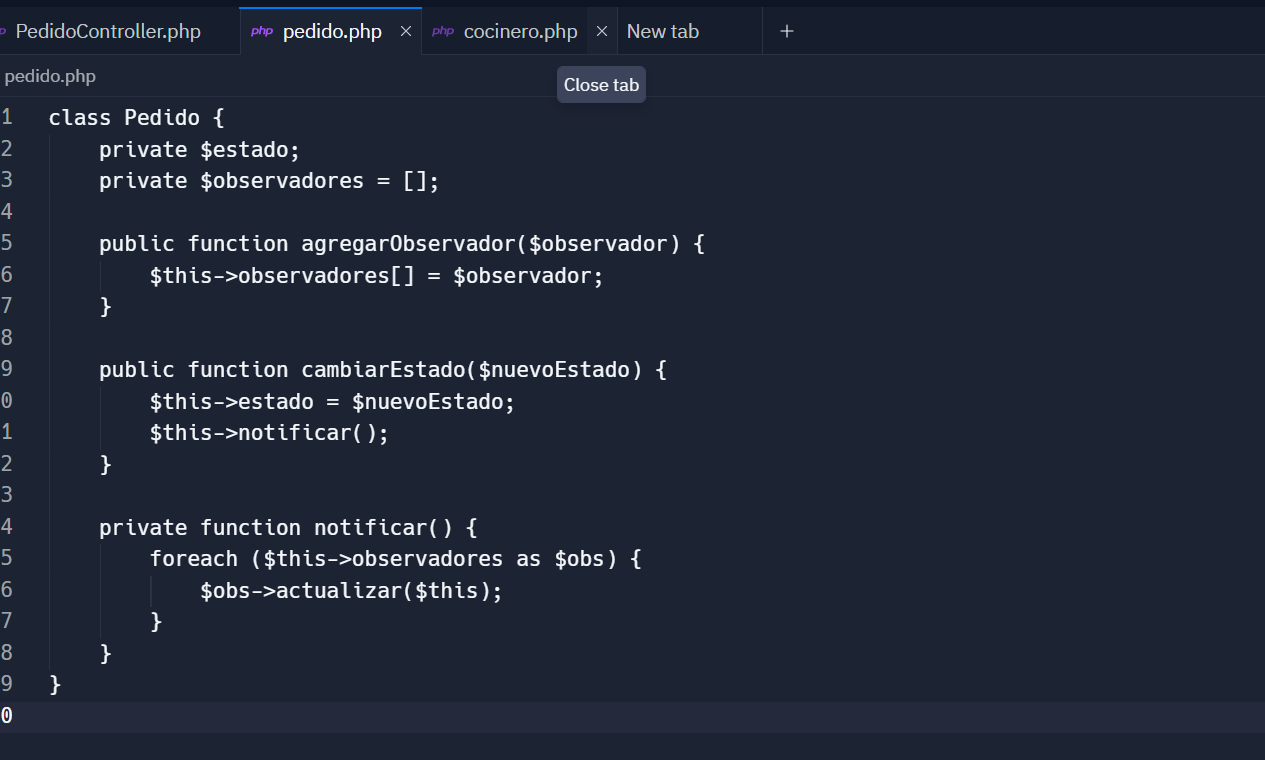


Este fragmento demuestra el uso de MVC e Inyección de Dependencias en el constructor.

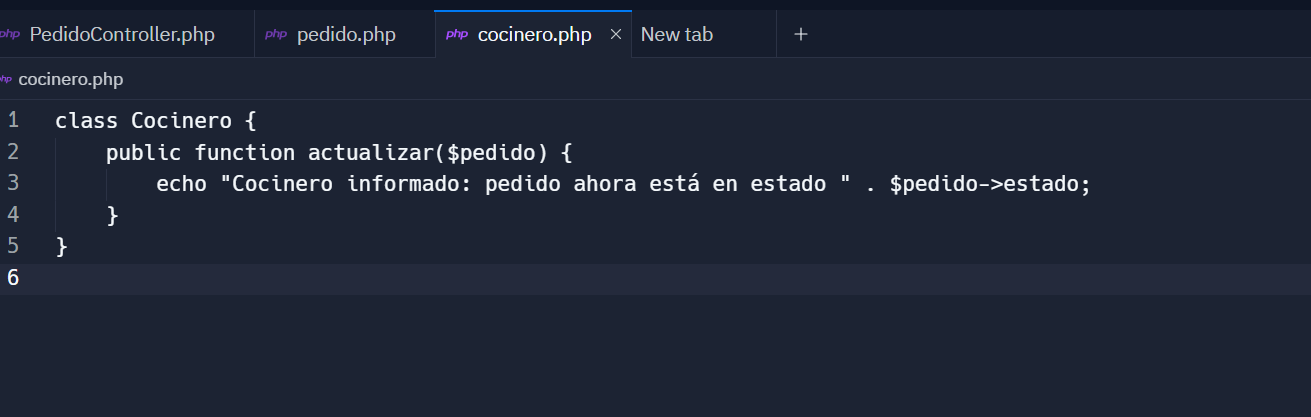
**Observer (Observador)**

El patrón Observer permite que cocinero1 y administrador1 se mantengan informados sobre cambios en pedido1.

Ejemplo (pedido notifica a observadores):

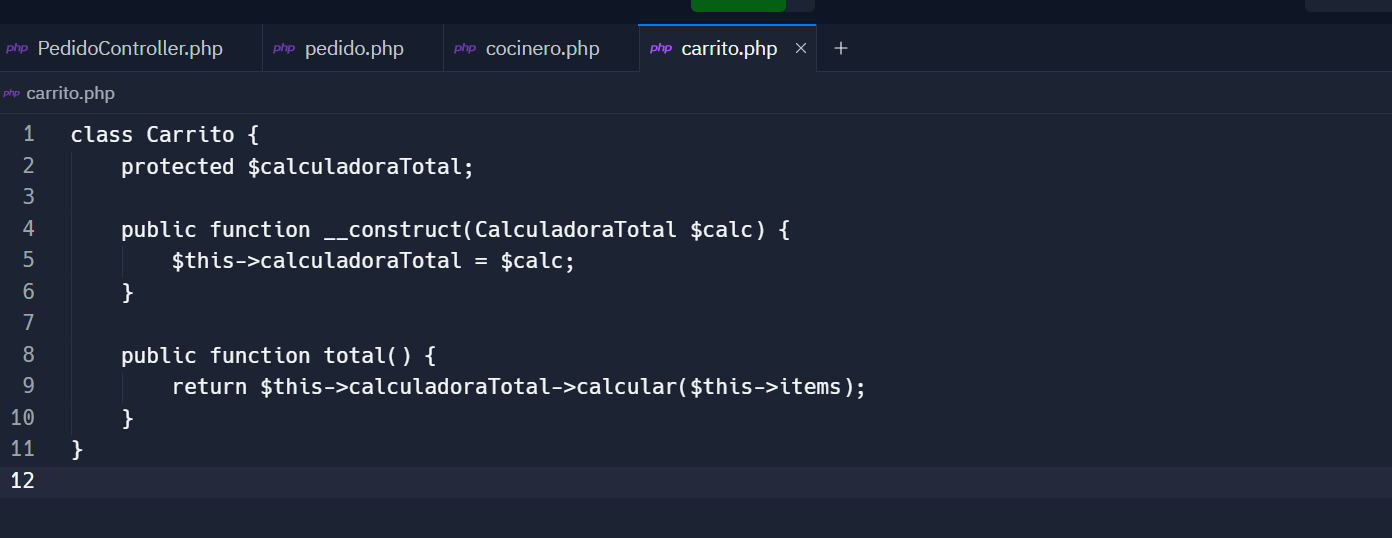


Y un ejemplo de observador:



**Inyección de Dependencias**

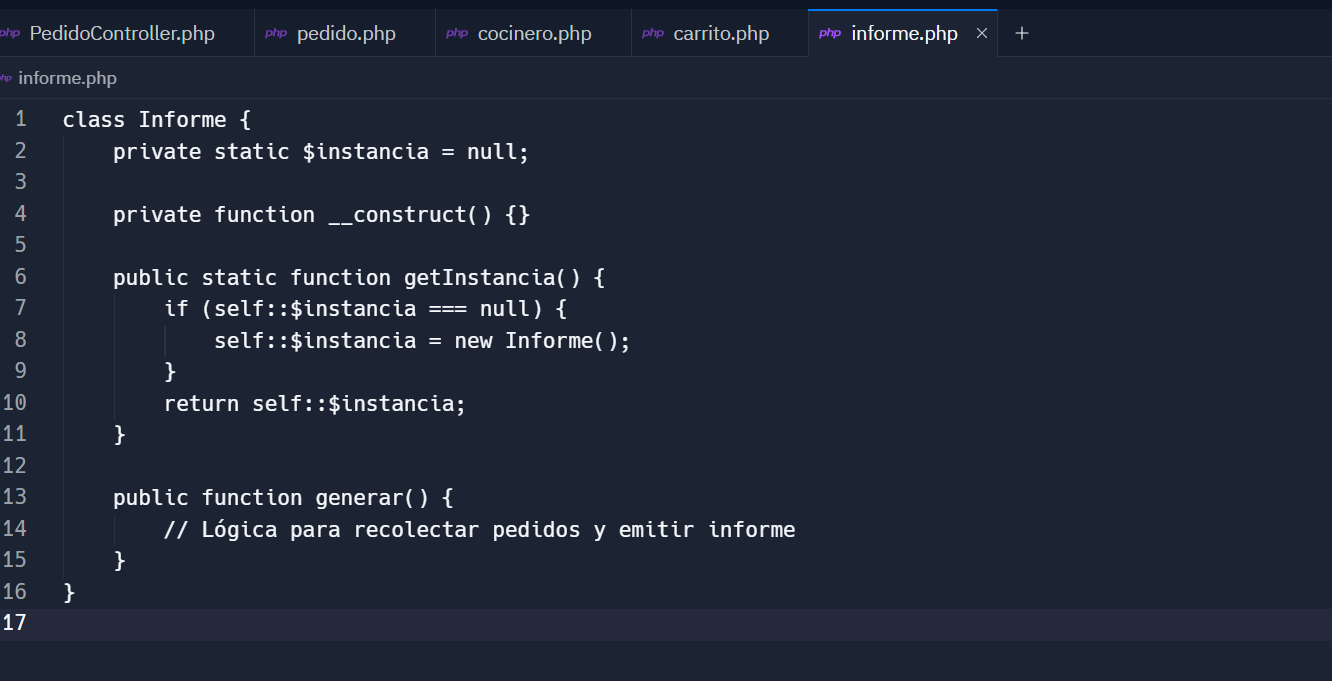
Ejemplo: carrito1 requiere un servicio de cálculo de totales:



Esto permite cambiar la implementación de cálculo sin modificar la clase Carrito.

**Singleton (ej. informe1)**

La clase informe1 es instanciada una única vez para garantizar consistencia en los reportes generados.



Esto asegura que cualquier controlador o módulo que necesite generar un reporte acceda a la misma instancia compartida

5.7.3 Restricciones del punto de vista  
Esta sección define las reglas, notaciones y estructuras necesarias para garantizar que el diseño basado en patrones se mantenga coherente, correcto y aplicable durante la implementación. Incluye el uso de lenguajes estructurados, restricciones arquitectónicas, y el diagrama de estructura compuesta como lenguaje de representación.

—

Lenguaje de descripción utilizado

Para describir las restricciones y relaciones entre entidades, se utilizaron dos lenguajes complementarios:

- Lenguaje Unificado de Modelado (UML), especialmente el diagrama de estructura compuesta, para representar visualmente los roles, conectores y colaboraciones entre clases como cliente1, carrito1, pedido1, cocinero1, informe1, etc.

- Lenguaje estructurado (fragmentos de código en PHP estilo Laravel), para mostrar cómo se aplican e implementan los patrones (MVC, Observer, Singleton, Inyección de Dependencias) dentro de las clases del sistema.

—

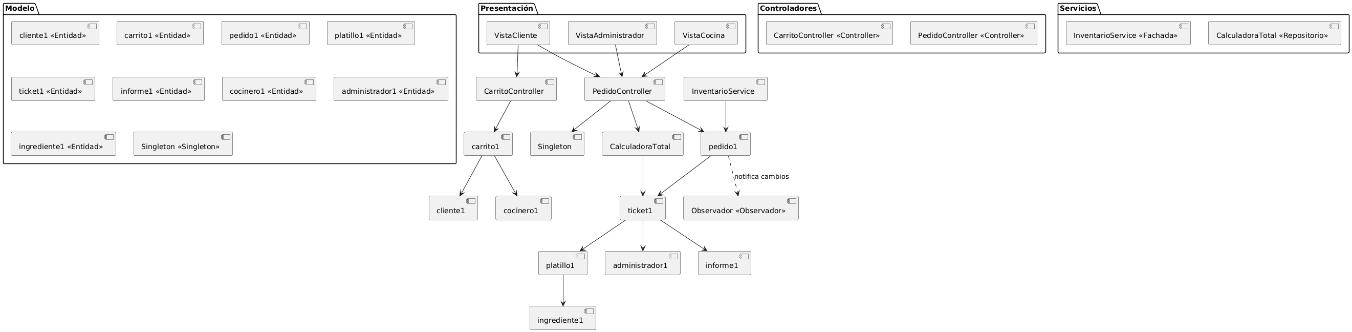
Diagrama de estructura compuesta (estructura interna de componentes)

Este diagrama representa cómo interactúan las clases internamente en el sistema, visualizando los patrones aplicados:

* cliente1 conecta con carrito1
* carrito1 se comunica con platillo1
* carrito1 se transforma en pedido1
* pedido1 notifica a cocinero1 y administrador1 (Observer)
* pedido1 se asocia con ticket1 e informe1 (Singleton)
* servicios como InventarioService y CalculadoraTotal son inyectados a pedido1 y carrito1 respectivamente (Inyección de dependencias)

 Restricciones arquitectónicas

Estas restricciones definen las reglas obligatorias que deben cumplirse en el diseño:

1. Separación de responsabilidades (MVC):

* Las clases cliente1, carrito1, pedido1, etc., no deben contener lógica de presentación.
* Las vistas no deben procesar lógica de negocio ni manipular datos directamente.
* Los controladores (como PedidoController) solo coordinan, no acceden directamente a la base de datos.

2. Control de instancias (Singleton):

* informe1 debe implementarse como clase Singleton. Solo puede existir una instancia en ejecución.
* ticket1 se crea por cada pedido, pero no debe tener múltiples asociaciones inversas hacia pedido1.

3. Notificación (Observer):

* pedido1 debe permitir el registro de observadores (cocinero1, administrador1) para recibir cambios de estado.
* Los observadores deben implementar una interfaz común (por ejemplo, ObservadorEstadoPedido) y registrarse correctamente.

4. Inyección de dependencias:

* Las clases carrito1, pedido1 y servicios auxiliares deben recibir sus dependencias desde afuera (por constructor o contenedor), no instanciarlas internamente.
* Ejemplo: PedidoService no debe crear una instancia de InventarioService; debe recibirla ya construida.

5. Restricciones de colaboración:

* cliente1 solo puede tener un carrito1 activo.
* carrito1 debe tener al menos un platillo1 para poder convertirse en pedido1.
* pedido1 no puede generar ticket1 si no ha pasado por los estados “confirmado” y “preparado”.
* informe1 solo se puede generar para pedidos entregados.

 Reglas complementarias de diseño:

* Los conectores deben mantener baja dependencia entre módulos.
* Las clases deben seguir el principio de responsabilidad única.
* Se deben evitar referencias cíclicas entre entidades (por ejemplo, carrito1 → pedido1 → carrito1 no debe existir).