

# DDD Factura Fácil.

## Informe de Modelado DDD: Context Map de FACTURA FÁCIL

### 1. Propósito del proyecto

Factura Fácil es una plataforma SaaS cuya misión principal es ofrecer a las micro y pequeñas empresas peruanas una solución de emisión de comprobantes electrónicos de manera ágil, eficiente y con la menor curva de aprendizaje. Se enfoca en:

- **Simplicidad y velocidad:** flujos guiados que permiten generar un comprobante en pocos pasos, minimizando errores de captura (autocompletear RUC/DNI, validaciones en línea).
- **Cumplimiento automático con SUNAT:** generación de XML, firma digital y envío automático a SUNAT, con descarga de XML/CDR.
- **Mínima curva de aprendizaje:** interfaz intuitiva que reduce la necesidad de capacitación extensa, brindando formularios autoexplicativos y mensajes de validación claros.
- **Visibilidad en tiempo real:** dashboards y reportes que muestran indicadores clave de ventas, clientes y productos para que el usuario tome decisiones rápidas e informadas.
- **Modularidad basada en DDD:** estructura de bounded contexts que separa responsabilidades (comprobantes, caja, indicadores, configuración y administración de precios), facilitando la evolución y el mantenimiento.

El objetivo es que una pequeña empresa, sin experiencia técnica, emita comprobantes y gestione su caja y reportes de negocio en segundos, con total confianza en el cumplimiento tributario.

### 2. Público objetivo

#### Microempresas y pequeños emprendedores

#### Usuarios principales

- Dueños de negocio y emprendedores
- Cajeros y operadores del punto de venta

### 3. Necesidades clave a satisfacer

- **Eficiencia operativa:** flujos de venta sencillos que minimizan tiempos y permiten atender más clientes.
- **Cumplimiento tributario:** generación y envío correcto de comprobantes a SUNAT, validación y aceptación sin intervención técnica.
- **Gestión de caja simplificada:** registrar apertura y cierre de caja con reportes de ingresos/egresos por turno.
- **Visibilidad inmediata:** monitorear ventas en tiempo real mediante un dashboard con gráficos y rankings.
- **Emisión ágil de comprobantes electrónicos:** generación rápida de facturas, boletas, guías y notas.
- **Registro de apertura y cierre de caja:** reportes de ingresos/egresos por turno y usuario.
- **Gestión de clientes, productos y servicios:** alta/edición rápida sin complejidad.

### 4. Segmento y necesidades

Segmento	Características del Negocio	Necesidades Funcionales Clave	Puntos de Dolor Identificados
Comercio minorista (bodegas, minimarkets)	Alto volumen de ventas diarias; manejo simple de inventario que no se modela en esta fase; enfoque en ventas rápidas.	<ul style="list-style-type: none"> <li>Emisión de boletas/facturas en segundos.- Validación automática de RUC/DNI.- Control de caja por turno.- Reportes de ventas en tiempo real.</li> </ul>	<ul style="list-style-type: none"> <li>Demoras al facturar clientes frecuentes.- Errores de captura de datos.- Dificultad para conciliar caja al hacerlo manualmente.</li> </ul>
Pequeños restaurantes/cafeterías	Ventas por ticket rápido, productos o servicios recurrentes; menú cambiante; caja por turno.	<ul style="list-style-type: none"> <li>Emisión de comprobantes en formato ticket y/o A4, A5.</li> <li>Arqueo diario de caja.- Ranking de productos (platos) más vendidos.- Ticket promedio por turno.</li> </ul>	<ul style="list-style-type: none"> <li>Confusión entre tipos de comprobante.</li> <li>Errores en cálculos de totales e IGV.</li> <li>Falta de métricas por turno/categoría.</li> </ul>
Servicios profesionales (talleres, freelancers, salones)	Facturación por servicios puntuales; clientes a crédito; seguimiento de cobros.	<ul style="list-style-type: none"> <li>Emisión fácil de facturas y boletas.</li> <li>Notas de crédito/débito para ajustes.</li> <li>Historial de clientes con pagos.</li> <li>Reporte de cuentas por cobrar.</li> </ul>	<ul style="list-style-type: none"> <li>Pérdida de facturas pendientes de pago.</li> <li>Carencia de recordatorios de cobro.</li> </ul>
Alquiler de inmuebles	Ingresos recurrentes mensuales; contratos a largo plazo; facturación periódica establecida.	<ul style="list-style-type: none"> <li>Creación automática de facturas recurrentes.</li> <li>Alerta de vencimiento de alquiler.</li> <li>Reportes mensuales de ingresos.</li> </ul>	<ul style="list-style-type: none"> <li>Emisión manual de facturas cada mes.</li> <li>Possible omisión de facturas.</li> <li>No hay alertas automáticas de vencimiento.</li> </ul>

## 5. Arquitectura SaaS y dominios relacionados ↗

Dentro de nuestra plataforma SaaS empresarial, Factura Fácil se encuentra en análisis como un nuevo dominio funcional especializado. Su integración considera los dominios ya existentes:

Dominio	Estado	Rol dentro de la arquitectura
<b>Factura Fácil</b>	En análisis	Servicio de facturación electrónica simplificada.
<b>Identidad y Autenticación</b>	Existente	Infraestructura central de acceso: gestiona el registro de usuario y contraseña para permitir el acceso a la plataforma.
<b>API Service de Facturación</b>	En análisis	Gestiona la creación, firmado, envío y almacenamiento de comprobantes electrónicos, así como recepción de CDR.

## 6. Relaciones de Factura Fácil con otros dominios ↗

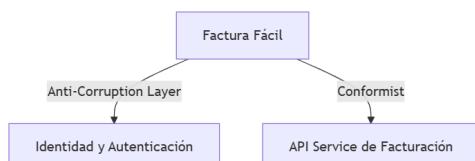
### 1. Identidad y Autenticación → Factura Fácil

- **Patrón:** Anti-Corruption Layer
- **Explicación:**
  - El usuario se registra (usuario/contraseña) en el dominio de Identidad y Autenticación para obtener acceso a Factura Fácil.
  - Factura Fácil consume únicamente la funcionalidad de registro y login; no gestiona roles ni permisos aquí.
  - Un adaptador ACL traduce esa información de acceso a un objeto interno **UsuarioRegistrado**, aislando el modelo de Factura Fácil de la estructura externa de Identidad y Autenticación.

### 2. API Service de Facturación → Factura Fácil

- **Patrón:** Conformist
- **Explicación:**
  - Al generar un comprobante, Factura Fácil envía el JSON/XML al API Service, el cual devuelve el XML firmado y el CDR.
  - Factura Fácil adopta esa respuesta “tal cual” sin crear un modelo intermedio, respetando el contrato externo definido por SUNAT.

## 7. Context Map desde perspectiva Factura Fácil ↗



## 8. Conclusiones

- Factura Fácil actúa como el dominio central para emisión de comprobantes y gestión de caja, mientras que el dominio de Identidad y Autenticación se limita a registrar usuarios y autenticar su acceso.
- La integración con Identidad y Autenticación utiliza el patrón Anti-Corruption Layer para aislar el modelo interno de Factura Fácil de la estructura de acceso externo.
- La integración con API Service de Facturación sigue el patrón Conformist, adoptando directamente el contrato de JSON/XML de SUNAT sin transformaciones.
- Esta separación clara de responsabilidades y patrones de integración garantiza que Factura Fácil pueda evolucionar y mantenerse sin verse afectada por cambios en los otros dominios.

## 9. Glosario de términos y conceptos de DDD

- **Dominio:** Lógica y conceptos de negocio de alto nivel.
- **Bounded Context:** Límite donde un modelo es consistente y posee un lenguaje ubicuo.
- **Anti-Corruption Layer (ACL):** Capa que traduce datos de un dominio externo al modelo interno, evitando contaminaciones.
- **Conformist:** Patrón en el cual se adopta sin cambios el modelo de un dominio externo para simplificar la integración.
- **Entidad (Entity):** Objeto de dominio con identidad única que persiste en el tiempo.
- **Objeto de Valor (Value Object):** Objeto inmutable definido por sus atributos; se compara por valor en lugar de identidad.
- **Agregado (Aggregate):** Conjunto de entidades y objetos de valor con una raíz que mantiene invariantes de consistencia.
- **Repositorio (Repository):** Interfaz que abstrae la persistencia y recuperación de agregados.
- **Servicio de Dominio (Domain Service):** Contiene lógica de negocio que no encaja en una sola entidad o agregado.
- **Evento de Dominio (Domain Event):** Mensaje que anuncia que algo relevante ha ocurrido en el modelo de dominio.
- **Specification:** Patrón para encapsular criterios de validación o búsqueda reutilizables.

# Dominio: Factura Fácil

## 1. Propósito del dominio [🔗](#)

El dominio **Factura Fácil** agrupa toda la lógica y los conceptos necesarios para que las micro y pequeñas empresas peruanas puedan generar, validar y gestionar comprobantes de pago electrónicos (facturas, boletas, notas de crédito/débito, guías de remisión electrónicas) de manera ágil, confiable y sin necesidad de conocimientos técnicos. Además, engloba la administración de clientes, productos/servicios, precios, caja y reportes, para ofrecer un sistema integral de gestión de venta y facturación en un único dominio.

## 2. Bounded Context del dominio [🔗](#)

Cada bounded context encapsula un subconjunto coherente de responsabilidad dentro del dominio. A continuación, se describen los principales contexts:

Tipo de Subdominio	Subdominio	Bounded Context	Propósito
<b>Core Domain</b>	Emisión Electrónica de Comprobantes	ComprobantesElectrónicosBC	Generar y enviar a SUNAT todos los comprobantes electrónicos (Facturas, Boletas, Notas de Crédito/Débito, Guías de Remisión).
<b>Supporting</b>	Gestión de Caja	ControlCajaBC	Registrar apertura y cierre de turnos de caja, movimientos (ingresos, egresos) y conciliar cada comprobante con su movimiento de caja correspondiente.
<b>Supporting</b>	Reporte de ventas	IndicadoresNegocioBC	Construir y mantener dashboards e informes en tiempo real: gráficos de ventas, ranking de productos, ranking de clientes, ticket

			promedio, actividad por vendedor, categorías más vendidas y exportación de reportes.
<b>Supporting</b>	Configuración del Sistema	ConfiguracionSistemaBC	Administrar parámetros globales: datos de la empresa (logo, RUC, razón social), series y correlativos para documentos, plantillas de impresión (PDF/ticket), formas de pago, gestión de empleados (usuarios, roles y permisos).
<b>Supporting</b>	Administración de Precios	ListaPreciosBC	Gestionar listas de precios (hasta 10 por producto/servicio), definir afectación de IGV (grava, exonerado, inafecto, detacciones), conversión de unidades y atributos contables (cuentas, centros de costo, presupuesto).
<b>Supporting</b>	Gestión de Clientes	GestionClientesBC	Crear y editar clientes, almacenar datos fiscales (razón social, nombres, correo, celular, forma de pago, moneda, género), adjuntar documentos, habilitar/deshabilita

			r clientes, y registrar historial de operaciones.
<b>Supporting</b>	Catálogo de Artículos (Productos/Servicios)	CatalogoArticulosBC	Gestionar ítems (SKU, descripción, tipo, marca, familia, categoría, subcategoría), afectación IGV, lotes/series, unidad de medida, almacén, campos contables, códigos de barras/SUNAT, estado activo/inactivo, variantes y combos, multimedia.

### 3. Glosario central del lenguaje ubicuo del dominio ↗

A continuación, los términos clave que se usan en todo el dominio Factura Fácil, con su definición funcional, uso en el sistema y en qué bounded contexts aparecen.

Término	Definición Funcional	Uso en el Sistema	Relación entre BC
<b>Comprobante</b>	Documento electrónico válido ante SUNAT que evidencia una operación de venta o servicio. Incluye Factura, Boleta, Nota de Crédito, Nota de Débito, y Guías de Remisión.	Se crea en <b>ComprobantesElectrónicosBC</b> , se envía al PSE/OSE mediante JSON/XML, y se almacena el XML + CDR.	Agregado principal en <b>ComprobantesElectrónicosBC</b> . Se vincula con <b>ControlCajaBC</b> al generar un movimiento de caja.
<b>Factura</b>	Comprobante que otorga crédito fiscal de IGV a empresas o entidades con RUC.	Subtipo de comprobante en <b>ComprobantesElectrónicosBC</b> ; contiene datos de RUC emisor/receptor,	Relaciona con <b>GestionCliente</b> sBC (cliente receptor). Afecta caja en <b>ControlCajaBC</b> .

		ítems con IGV y totales.	
<b>Boleta</b>	Comprobante para venta a consumidor final (persona natural), sin crédito fiscal.	Subtipo en <b>ComprobantesEl ectrónicosBC</b> ; similar a Factura pero sin RUC obligatorio.	Relaciona con <b>GestionCliente sBC</b> (opcional). Afecta caja en <b>ControlCajaBC</b> .
<b>Nota de Crédito</b>	Documento que anula o reduce el valor de un comprobante anterior por devoluciones o correcciones.	Generada en <b>ComprobantesEl ectrónicosBC</b> ; referencia el comprobante original (serie y correlativo).	Impacta métricas en <b>IndicadoresNeg ocioBC</b> y saldo en <b>ControlCajaBC</b> .
<b>Nota de Débito</b>	Documento que incrementa el valor de un comprobante previo por cargos adicionales.	Subtipo en <b>ComprobantesEl ectrónicosBC</b> ; referencia el comprobante original y contiene montos adicionales con su IGV.	Impacta métricas en <b>IndicadoresNeg ocioBC</b> y saldo en <b>ControlCajaBC</b> .
<b>Guía de Remisión</b>	Documento que acompaña el traslado de bienes y describe remitente, destinatario e ítems transportados.	En <b>ComprobantesEl ectrónicosBC</b> ; puede crearse independiente o vincularse a una Factura/Boleta.	No afecta caja, pero se relaciona con inventario fuera de este dominio (no modelado aquí).
<b>CDR (Constancia de Recepción)</b>	Confirmación de SUNAT que indica aceptación o rechazo de un comprobante electrónico.	Generado por PSE o SUNAT al recibir el XML; se almacena en <b>ComprobantesEl ectrónicosBC</b> para representar el estado (aceptado o rechazado).	Sirve para actualizar el estado del comprobante en <b>ComprobantesEl ectrónicosBC</b> .
<b>Turno de Caja</b>	Período de tiempo en que un usuario	Agregado en <b>ControlCajaBC</b> ;	Relaciona movimientos de

	opera la caja registradora, con saldo inicial y final.	registra <code>usuarioId</code> , <code>fechaApertura</code> , <code>fechaCierre</code> , <code>saldoInicial</code> , <code>saldoFinal</code> .	caja con los eventos de <code>ComprobantesElectrónicosBC</code> .
<b>Movimiento de Caja</b>	Registro de un ingreso (venta, anticipo) o egreso (devolución, gasto) asociado a un Turno de Caja.	Entidad en <code>ControlCajaBC</code> : campos <code>fecha</code> , <code>tipo</code> ( <code>ingreso/egreso</code> ), <code>monto</code> , <code>descripcion</code> .	Se crea cuando en <code>ComprobantesElectrónicosBC</code> se emite un comprobante pagado en efectivo, o cuando el usuario registra manualmente.
<b>Dashboard</b>	Interfaz que muestra métricas clave de negocio en tiempo real: ventas, ranking de productos, ranking de clientes, ticket promedio, actividad por vendedor.	Construido en <code>IndicadoresNegocioBC</code> ; consume eventos “ComprobanteEmitido” y “ComprobanteAnulado” de <code>ComprobantesElectrónicosBC</code> para actualizar contadores y rankings.	Visualiza datos agregados de <code>ComprobantesElectrónicosBC</code> , <code>GestionClientesBC</code> , <code>CatalogoArticulosBC</code> , <code>ControlCajaBC</code> .
<b>Lista de Precios</b>	Conjunto de precios para un producto o servicio, que pueden variar por canal, cliente o promoción.	Entidad en <code>ListaPreciosBC</code> : contiene <code>productId</code> , <code>precioBase</code> , <code>impuestoAplicable</code> y <code>fechas de vigencia</code> .	Consumida por <code>ComprobantesElectrónicosBC</code> al calcular totales e impuestos.
<b>Serie y Correlativo</b>	Prefijo y número consecutivo autorizado por SUNAT para identificar cada	Objeto de valor en <code>ConfiguracionParametrosSistemaBC</code> ; asignado a cada comprobante emitido en	Usado también en <code>ListaPreciosBC</code> para diferenciar series por sucursal o canal.

	comprobante (ej. "F001-00001234").	<b>ComprobantesEl ectrónicosBC .</b>	
<b>Afectación IGV</b>	Clasificación tributaria de un ítem (grava, exonerado, inafecto, detacción) que determina cálculo de IGV.	Campo en la entidad <b>ÍtemComprobant e dentro de ComprobantesEl ectrónicosBC ;</b> influye en la suma de impuestos del comprobante.	Definido en <b>CatalogoArticu losBC y consumido en ComprobantesEl ectrónicosBC al generar el comprobante.</b>
<b>Cliente</b>	Persona natural o jurídica que recibe el comprobante; identificada por DNI (clientes de boleta) o RUC (clientes de factura) y otros tipos de documento.	Entidad en <b>GestionCliente sBC : campos clienteId , tipoDocumento (DNI/RUC) , nombreRazónSoc ial , direcciónFisca l , correo , celular , formaPago , moneda , genero , estado (activo/inacti vo) , adjuntos , notas .</b>	Referenciado por <b>ComprobantesEl ectrónicosBC</b> (al emitir un comprobante) y por <b>IndicadoresNeg ocioBC</b> (ranking de clientes por saldos).
<b>Producto / Servicio</b>	Ítem que se vende o presta: puede ser un bien físico o un servicio; caracterizado por SKU, descripción, categorías, afectación tributaria, precio, etc.	Entidad en <b>CatalogoArticu losBC : campos productoId , SKU , nombre , tipo (bien/servicio ) , marca , familia , categoría , subcategoría , afectaciónIGV , unidadMedida ,</b>	Consumido por <b>ComprobantesEl ectrónicosBC</b> (al agregar ítems al comprobante), por <b>IndicadoresNeg ocioBC</b> (ranking de productos).

```
almacén , peso ,  
camposContable  
s , estado ,  
atributosAdici  
onales ,  
códigoBarras ,  
códigosSUNAT ,  
multimedia .
```

#### Observaciones

- En Gestión Clientes BC, además de los datos fiscales, se incluyen adjuntos (fotos, documentos, archivos, comentarios), el historial de operaciones y estado de habilitación.
- En Catálogo Artículos BC, se modelan todos los atributos de ítems descritos (características, códigos, precios, unidades, etc.).
- El glosario refleja el lenguaje ubicuo utilizado en todos los bounded contexts, de modo que todo el equipo comparta la misma semántica.

# ControlCajaBC - Antonio H

## 1. Propósito

ControlCajaBC gestiona todo lo relacionado con los turnos de caja en Factura Fácil: apertura, cierre, movimientos de ingresos/egresos y la conciliación automática de comprobantes electrónicos con el efectivo. Su misión es garantizar la trazabilidad del dinero, el control de fondos y la consistencia de saldos en cada punto de venta, proporcionando información clave para reportes y auditorías financieras. En resumen, este contexto asegura que el manejo de la caja (efectivo) esté sincronizado con la emisión de comprobantes, brindando confianza en el cierre diario y en el cumplimiento tributario.

## 2. Responsabilidades Clave

Las responsabilidades principales de ControlCajaBC se enumeran a continuación, con sus respectivas descripciones:

Responsabilidad	Descripción
<b>Apertura de Turno</b>	Registrar el inicio de un turno de caja por usuario, indicando <b>saldo inicial</b> , hora de inicio y responsable asignado.
<b>Registro de Movimientos</b>	Registrar <b>ingresos</b> y <b>egresos</b> de caja – ya sean automáticos (derivados de eventos de venta) o manuales – incluyendo el motivo, y referencia a un comprobante (venta) o descripción de gasto/retirada.
<b>Conciliación de Comprobantes</b>	Asociar cada comprobante electrónico emitido con su correspondiente movimiento de caja, identificando discrepancias entre ventas registradas y efectivo disponible. Esto garantiza que por cada factura/boleta haya un ingreso en caja, y viceversa.
<b>Cálculo de Saldos</b>	Calcular en todo momento el <b>saldo actual</b> , el <b>saldo final esperado</b> y la <b>diferencia de arqueo</b> (desfase) al cerrar el turno. Estas métricas permiten alertar de inconsistencias (faltantes o sobrantes de dinero).
<b>Cierre de Turno</b>	Finalizar formalmente un turno de caja, aplicando validaciones de discrepancia (dentro de tolerancia) y generando el <b>reporte de cierre</b> . Implica realizar el

	conteo de efectivo físico, comparar contra saldo final calculado y bloquear el cierre si hay diferencias excesivas.
<b>Publicación de Eventos</b>	Emitir <b>eventos de dominio</b> relevantes al resto del sistema: por ejemplo, <code>TurnoCajaAbierto</code> , <code>MovimientoCajaRegistrado</code> y <code>TurnoCajaCerrado</code> . Estos eventos alimentan auditorías internas y la sincronización con módulos de reportería e indicadores.

### 3. Contexto y Motivación

En micro y pequeñas empresas es común llevar la caja de manera informal, lo que conlleva errores, descuadres y potenciales fraudes. **ControlCajaBC existe para abordar estos problemas:**

- **Automatización:** Automatiza la apertura y cierre de caja, reduciendo el tiempo y la carga operativa manual. Un proceso guiado disminuye la probabilidad de omisiones (ej. olvido de cerrar caja al final del día).
- **Conciliación ventas-caja:** Asegura que cada comprobante electrónico (factura/boleto) tenga su reflejo en caja, eliminando discrepancias entre las ventas registradas y el dinero disponible. Esto previene pérdidas no detectadas y facilita el cuadro diario.
- **Visibilidad en tiempo real:** Provee visibilidad inmediata del estado de la caja (**saldo actual, ingresos, egresos, diferencias**) para gerentes y contadores. Esto permite tomar decisiones rápidas ante anomalías (por ejemplo, si falta dinero o un cajero excede cierto monto).
- **Auditoría y cumplimiento:** Facilita auditorías internas/externas mediante un historial de eventos detallados (aperturas, movimientos, cierres) y reportes formales. Esto contribuye al cumplimiento de políticas contables y fiscales, ya que toda transacción queda registrada y asociada a un responsable.
- **Manejo de movimientos no vinculados:** Permite gestionar movimientos de caja que no provienen directamente de una venta (por ejemplo, gastos de caja chica, retiros de efectivo), integrándolos en el mismo control para un cuadro completo de la caja.

ControlCajaBC surge de la necesidad de **reducir errores y fraudes en la gestión de efectivo**, asegurando que el dinero en caja cuadre con las ventas registradas, con la mínima intervención manual y máxima transparencia.

### 4. Fuentes de Información

Fuente (BC)	Datos que Provee
ComprobantesElectonicosBC	ControlCajaBC consume eventos del contexto ComprobantesElectonicosBC. Específicamente, <code>ComprobanteEmitido</code> y <code>ComprobanteAnulado</code> son eventos externos que aportan datos (ID del comprobante, monto total, tipo de pago, fecha, motivo de anulación, etc.) para generar automáticamente ingresos o revertir

	<p>movimientos en caja. Esto asegura que las ventas registradas en Factura Fácil (boletas, facturas) impacten inmediatamente la caja.</p>
ConfiguracionSistemaBC	<p>A través de ConfiguracionSistemaBC, el diseño de ControlCajaBC se basa en parámetros configurables como la <b>moneda</b> de operación, formatos de numeración, horarios de operación y especialmente la <b>tolerancia de redondeo o de diferencia de caja permitida</b>. Además, este contexto emite eventos como <b>PermisosUsuarioActualizado</b> que informan roles/permisos de usuarios, determinando si un cajero está habilitado para abrir/cerrar caja. Por ejemplo, si un usuario pierde el permiso de “AbrirCaja”, ControlCajaBC debe bloquear esa operación para dicho usuario.</p>
GestionClientesBC	<p>ControlCajaBC recibe información del contexto GestionClientesBC en eventos como <b>ClienteActualizado</b>, que trae datos de estado de cliente, límites de crédito y forma de pago. Esto influye en validaciones al registrar ingresos a cuenta (por ejemplo, ventas a crédito que no implican entrada inmediata de efectivo). También, en escenarios de pagos externos (como integración con pagos en línea), podría recibir notificaciones para registrar ingresos correspondientes (ej. un evento <b>PagoExternoRecibido</b>).</p>
Stakeholders (usuarios clave)	<p>Dueños de negocio y contadores solicitaron visibilidad clara del efectivo en caja versus ventas, así como alertas ante <b>faltantes</b> de dinero. Sus experiencias con cierres de caja manuales (libros o Excel) guiaron la necesidad de funcionalidades como reportes automáticos, tolerancias de cierre y delegación de cierre a supervisores.</p>

Normativas contables y tributarias	Se consideraron lineamientos de control de caja y arqueo diarios comúnmente recomendados por prácticas contables. Si bien SUNAT (ente tributario peruano) no exige un módulo de caja, las empresas deben poder mostrar consistencia entre sus ventas (comprobantes) y sus ingresos de efectivo para fines fiscales y de auditoría. Esto motivó reglas como conciliación obligatoria de comprobantes y registro de diferencias permitidas.
Documentación y experiencia previa	Se revisaron manuales de cajas de pequeños comercios y recomendaciones de software POS (Point of Sale) para microempresas. Estas fuentes inspiraron funcionalidades como la <b>delegación de cierre</b> (un supervisor puede cerrar la caja iniciada por otro cajero en caso de ausencia) y el manejo de <b>ajustes por discrepancia</b> (registro de un movimiento de ajuste cuando hay diferencias fuera de tolerancia, manteniendo registro claro de quién autorizó el ajuste).

En resumen, ControlCajaBC se diseña apoyándose en la **legislación tributaria** (para asegurar consistencia ventas-efectivo), en **necesidades operativas** manifestadas por los usuarios (dueños, cajeros, contadores) y en la **integración con otros sistemas internos** (eventos de ventas, clientes, configuración) que proveen los datos necesarios para que la caja opere con mínima intervención manual.

## 5. Modelo de Dominio

**Visión General:** El modelo de dominio de ControlCajaBC está centrado en el **agregado raíz TurnoCaja**, que encapsula la apertura, los movimientos y el cierre de caja, garantizando las invariantes de consistencia a lo largo de ese ciclo. Este agregado contiene entidades para representar cada movimiento de dinero y el responsable, así como objetos de valor para detalles inmutables (códigos, montos, etc.). A continuación se describen sus principales elementos:

### 5.1 Agregado Raíz

Entidad Raíz	Descripción
TurnoCaja	Representa una <b>sesión de caja</b> abierta por un usuario (cajero) en un determinado punto de venta. Es la entidad raíz que agrupa y gobierna a los movimientos de efectivo realizados en ese turno. Mantiene <b>invariantes</b> como: solo puede haber un turno abierto por estación a la vez, no se puede cerrar un turno sin haberlo abierto, y no se pueden registrar movimientos cuando el turno está cerrado. El <b>ciclo de vida</b> de un TurnoCaja va de <b>ABIERTO</b> a <b>CERRADO</b> , y durante ese ciclo va acumulando movimientos y calculando saldos.

Entre sus **atributos principales** están: `turnoId` (identificador único del turno, p. ej. UUID), `usuarioId` o `responsableId` (identidad del cajero responsable), `codigoCaja` (identificador de la estación o caja física), `fechaHoraApertura` y `fechaHoraCierre`, `saldoInicial` (monto con el que inicia la caja), `saldoFinal` (monto calculado al cierre, teóricamente esperado) y `efectivoContado` (monto físico contado al cierre). También registra la `diferenciaCaja` entre el efectivo contado y el saldo final esperado, y un `estadoTurno` (enum **ABIERTO/CERRADO** que indica si el turno sigue activo). El agregado `TurnoCaja` es responsable de métodos como `abrirTurno(saldoInicial)`, `registrarMovimiento(...)` y `cerrarTurno(efectivoContado)` aplicando las reglas de negocio antes de cambiar de estado. Además, `TurnoCaja` **posee una relación 1:N** con sus movimientos (`MovimientoCaja`) y conoce a su **ResponsableCaja** (quien abrió el turno).

## 5.2 Entidades del Agregado

Entidad	Descripción
<code>MovimientoCaja</code>	Registra un ingreso o egreso asociado a un <code>TurnoCaja</code> , referenciando comprobantes o justificando retiros/gastos.
<code>ResponsableCaja</code>	Identifica al usuario (cajero) y estación de trabajo a cargo del <code>TurnoCaja</code> .

### **MovimientoCaja (Entidad del Agregado):**

Representa cada **transacción de efectivo** dentro de un turno: un **ingreso** (por ejemplo, una venta en efectivo) o un **egreso** (por ejemplo, un retiro para pago de gastos o un error que se corrige). Cada `MovimientoCaja` pertenece a un `TurnoCaja` (relación Muchos a Uno). Sus **atributos** incluyen `movimientoId` (UUID único del movimiento), `tipoMovimiento` (enumeración INGRESO o EGRESO), `origenMovimiento` (indicador de si proviene de un evento de negocio, como una venta, o es manual; e.g. `EVENTO_COMPROBANTE` vs `MANUAL`), `monto` (valor monetario positivo del movimiento), `descripcion` o referencia (ej. “Pago en efectivo Factura #001” o “Retiro para caja chica”). Puede opcionalmente relacionarse con un `comprobanteId` (ID del comprobante de venta asociado, si aplica) o con un `clienteId` (si el movimiento corresponde a un pago a cuenta de un cliente) para fines de trazabilidad. **Comportamiento:** al crear un `MovimientoCaja`, se valida que el monto sea positivo y, dependiendo del tipo, el `TurnoCaja` actualizará su saldo actual sumando o restando ese monto. Los movimientos automáticos (provenientes de eventos de venta) traen integridad: deben corresponder a un comprobante válido y no pueden alterarse manualmente (son inmutables). También existe la posibilidad de **anular** un movimiento (por ejemplo, revertir un ingreso si una venta es anulada): esto típicamente se modela creando un nuevo `MovimientoCaja` inverso (de tipo contrario) en lugar de borrar el original, manteniendo el historial.

### **ResponsableCaja (Entidad):**

Identifica al **usuario cajero** y la **estación de trabajo** asociada al `TurnoCaja`. En algunos diseños podría ser simplemente atributos dentro de `TurnoCaja` (`responsableId` y `codigoCaja`), pero aquí se le da entidad propia para agrupar lógica relacionada al responsable. Por ejemplo, `ResponsableCaja` puede contener la referencia al **perfil/rol del usuario** y verificar sus permisos al intentar abrir o cerrar el turno. Sus atributos básicos son `responsableId` (identificador del usuario, usualmente vinculado al contexto de Identidad y Autenticación) y `estacionId` o `codigoCaja` (identificador de la caja registradora o terminal físico utilizado). **Comportamiento:** puede tener operaciones para validar si el usuario tiene permiso para apertura o cierre (`validarPermisosApertura()` / `validarPermisosCierre()`), consultando una lista de roles/permisos que es actualizada vía eventos de `ConfiguracionSistemaABC` (ej. un evento de permisos actualizado cambiaría los atributos de permisos del `ResponsableCaja`). En esencia, `ResponsableCaja` aporta al `TurnoCaja`

la información de *quién* opera la caja y en *qué punto de venta*, influyendo en las reglas de negocio (por ejemplo, solo el mismo responsable o un supervisor autorizado puede cerrar el turno).

### 5.3 Objetos de Valor

ControlCajaBC utiliza **Value Objects (VO)** para modelar conceptos inmutables o compuestos que agregan claridad al dominio:

Objeto de Valor	Descripción
CodigoCaja	Valor <b>inmutable</b> que identifica de forma única una estación de caja o terminal. Es típicamente una cadena (ej: "POS-01", "CAJA-0001") asignada al abrir el turno y que no cambia durante todo el TurnoCaja. Se usa para garantizar que no se dupliquen turnos abiertos en la misma caja física, ya que actúa como clave de unicidad junto con el estado abierto.
Monto	Objeto de valor que representa una <b>cantidad monetaria</b> . Incluye el valor numérico con la precisión adecuada (centavos) y la <b>moneda</b> en la que opera (por ejemplo, PEN para soles peruanos). Garantiza invariantes como que un monto de movimiento debe ser <b>&gt; 0</b> , encapsulando validaciones de formato o precisión decimal permitida. Podría reutilizarse en todo el dominio para cantidades de dinero.
FechaHora	Sello de tiempo que registra momentos clave (apertura, cada movimiento, cierre) con precisión de fecha y hora <b>incluyendo la zona horaria</b> . Se utiliza un VO para asegurar formato consistente (por ejemplo, utilizar UTC o zona local explícita) y facilitar comparaciones/ordenamiento. Este VO es inmutable y se crea tomando la hora actual del sistema en el momento del evento.
TipoMovimiento	Enumeración que clasifica un movimiento de caja como <b>INGRESO</b> o <b>EGRESO</b> . Es un VO de tipo enum que se usa tanto para dar significado al signo del monto (un ingreso suma, un egreso resta) como para habilitar ciertas validaciones (por ejemplo, un egreso podría requerir un campo de descripción obligatoria).
SaldoCaja	Representa un <b>resumen de los saldos</b> de un turno de caja, usualmente calculado al vuelo. Puede considerarse un VO compuesto que agrupa varios valores relacionados al estado financiero del turno: por ejemplo, <code>saldoInicial</code> , <code>ingresosTotales</code> , <code>egresosTotales</code> , <code>saldoActual</code> ( <code>saldoInicial + ingresos - egresos</code> ), <code>efectivoContado</code>

(monto físico contado al cierre) y **diferenciaCaja** (**saldoActual - efectivoContado**). Como VO, **SaldoCaja** puede recalcularse cada vez que hay cambios y usarse para generar reportes o comparar contra tolerancias. Es inmutable en el sentido que se construye a partir de los movimientos al cerrar el turno (o en consultas intermedias) y si cambia algún movimiento, se construye un nuevo VO. Encapsular estos valores juntos asegura consistencia (por ejemplo, que la diferencia corresponda exactamente a la resta de **saldoActual** y **efectivoContado**).

Cada VO mencionado se define por sus atributos y reglas internas (ej. **CodigoCaja** quizás valida un formato específico, **Monto** maneja decimales según configuración regional, **TipoMovimiento** puede limitar valores a INGRESO/EGRESO, etc.). Al ser inmutables, brindan seguridad de que, una vez creados, los datos críticos (como montos y códigos) no se alterarán indebidamente durante el flujo.

### 5.3 Servicios de Dominio ↵

Dentro de ControlCajaBC, gran parte de la lógica reside en el propio agregado TurnoCaja. Sin embargo, hay ciertas operaciones complejas o **reglas de negocio transversales** que pueden implementarse como **Servicios de Dominio** (Domain Services) porque no pertenecen naturalmente a una sola entidad:

- **Servicio de ConciliaciónCaja:** Coordina la lógica para **conciliar comprobantes con la caja**. Por ejemplo, cuando llega un evento **ComprobanteEmitido** desde ComprobantesElectonicosBC, este servicio determina a qué turno aplicar el ingreso, creando el MovimientoCaja correspondiente vía el agregado TurnoCaja. Del mismo modo, si llega un **ComprobanteAnulado**, el servicio orquesta la creación de un movimiento de egreso (reverso) en el turno adecuado. Al centralizar esta lógica, se asegura que incluso si la emisión de comprobantes es asíncrona, la caja siempre reflejará correctamente esas operaciones.
- **Servicio de AutorizaciónCaja:** Encapsula reglas sobre **quién** puede realizar ciertas acciones de caja. Por ejemplo, podría exponer métodos como **puedeAbrirCaja(usuario)** o **puedeCerrarCaja(usuario, turno)** que internamente revisan el rol/permisos del usuario (posiblemente consultando datos provistos por ConfiguracionSistemaBC sobre ese usuario). Aunque la verificación de permisos ocurre a nivel de aplicación usualmente, tener un servicio de dominio para reglas complejas (ej: “solo el responsable original o un Supervisor con permiso especial puede cerrar el turno”) permite reutilizar esa lógica en distintos puntos (en validaciones previas al cierre, en la delegación de cierre, etc.).
- **Servicio de AjusteDiscrepancia:** Aplica la política cuando hay una diferencia de caja fuera de tolerancia. Por ejemplo, si al intentar cerrar un turno la **diferenciaCaja** excede el margen permitido, este servicio se encarga de marcar el turno con estado “Pendiente por ajuste” y quizás notificar a IndicadoresNegocioBC o a un módulo de alertas. También podría ejecutar un **proceso compensatorio**: por ejemplo, generar un registro (MovimientoCaja de ajuste) cuando un supervisor indica cómo se resolvió la diferencia (se encontró el faltante o sobrante, o se lo anota como pérdida/ganancia extraordinaria). La existencia de este servicio encapsula la lógica de qué hacer ante discrepancias (reglas RN-CB-004 y RN-CB-005, ver sección de Reglas), separando la decisión de la simple detección que hace el agregado.
- **ServicioReporteCaja:** Genera el **reporte de cierre de caja** (generalmente un PDF o documento resumido). Aunque la compilación del reporte involucra datos del TurnoCaja y sus movimientos (entidades de dominio), la generación de PDF en sí es infraestructura. Sin embargo, la **composición de la información** (qué datos incluir, cálculos como

total de ingresos, total de egresos, diferencias) puede considerarse lógica de dominio que este servicio prepara. Por ejemplo, un método `generarResumenCierre(turnoId)` podría recopilar del TurnoCaja los movimientos clasificados, totales, etc., y retornar un objeto de dominio `ReporteCierre` que luego la capa de aplicación formatea a PDF. De esta forma, se mantiene la lógica de compilación de métricas dentro del dominio (donde se entienden las reglas de negocio, como excluir ciertos movimientos o cómo presentar el cálculo de diferencia).

Estos servicios de dominio **no tienen estado propio** (son stateless); actúan sobre las entidades/VOs pasados como parámetros. Su existencia asegura que la lógica que involucra **múltiples agregados o dependencias externas** se agrupa claramente. Por ejemplo, validar un pago a crédito podría requerir datos del cliente (límite de crédito en GestiónClientesBC) y del estado de caja actual; un Domain Service podría orquestar esa validación combinada. Cada método de estos servicios define **contratos claros** dentro del dominio (ej.: `conciliarVenta(con Comprobante)` o `autorizarCierre(con TurnoCaja, usuario)`), facilitando su prueba unitaria y uso consistente en toda la aplicación.

## 6. Relaciones con Otros Bounded Contexts

Este apartado describe cómo **ControlCajaBC** interactúa con otros contexts para asegurar la consistencia operativa y alimentar la reportería en **IndicadoresNegocioBC**:

Relación	Función
<b>ComprobantesElectonicosBC → ControlCajaBC</b>	Los eventos <code>ComprobanteEmitido</code> y <code>ComprobanteAnulado</code> generan movimientos automáticos en caja.
<b>GestionClientesBC → ControlCajaBC</b>	El evento <code>ClienteActualizado</code> aporta estado, límites de crédito y forma de pago para validar ingresos.
<b>ConfiguracionSistemaBC → ControlCajaBC</b>	El evento <code>PermisosUsuarioActualizado</code> provee roles y permisos que habilitan o bloquean apertura y cierre.
<b>ControlCajaBC → IndicadoresNegocioBC</b>	Publica <code>TurnoCajaAbierto</code> , <code>MovimientoCajaRegistrado</code> y <code>TurnoCajaCerrado</code> para dashboards y métricas.

### 6.1 Consumo de Eventos (Inbound)

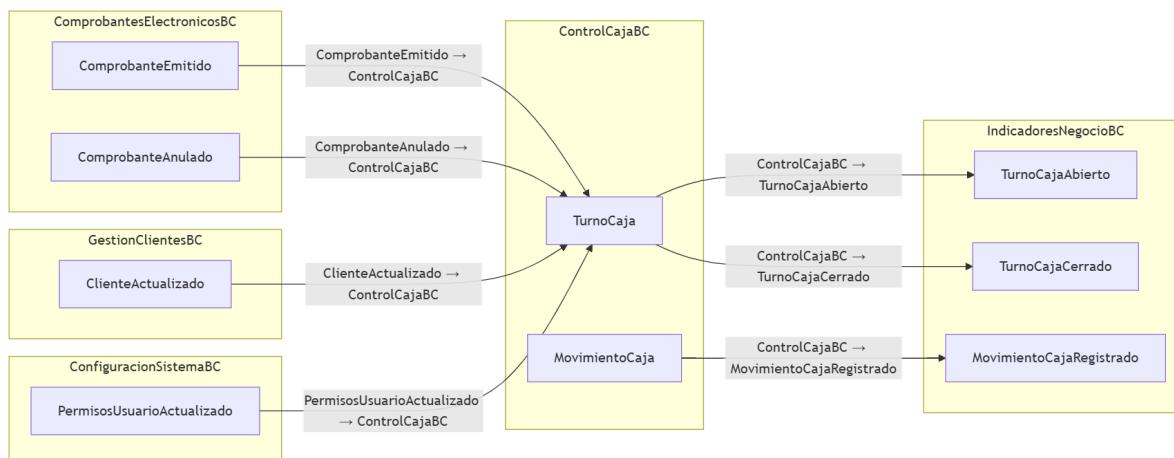
Fuente (BC)	Evento Consumido	Descripción del Dato
ComprobantesElectonicos BC	ComprobanteEmitido	ID, total, tipo de pago y fecha/hora para generar ingreso en caja.

ComprobantesElectonicosBC	ComprobanteAnulado	ID y motivo de anulación para revertir movimiento en caja.
GestionClientesBC	ClienteActualizado	Estado del cliente, límite de crédito y forma de pago.
ConfiguracionSistemaBC	PermisosUsuarioActualizado	Roles y permisos vigentes para apertura/cierre de caja.

## 6.2 Publicación de Eventos (Outbound)

Destino (BC)	Evento Publicado	Propósito / Uso
IndicadoresNegocioBC	TurnoCajaAbierto	Registrar inicio de turno en dashboards de ventas y flujo de caja.
IndicadoresNegocioBC	MovimientoCajaRegistrado	Actualizar métricas de ingresos/egresos y ticket promedio en reportería.
IndicadoresNegocioBC	TurnoCajaCerrado	Incluir cierre de turno en reportes de conciliación y alertas de diferencias.

## 6.3 Mapa de Contexto (Mermaid)



## 7. Restricciones y Reglas de Negocio

Restricción	Detalle

RES-01	Sólo un <code>TurnoCaja</code> puede estar abierto por estación de caja simultáneamente.
RES-02	No se pueden registrar <code>MovimientoCaja</code> sin un <code>TurnoCaja</code> abierto.
RES-03	Movimientos automáticos requieren un <code>comprobanteId</code> válido.
RES-04	Movimientos automáticos son inmutables y no pueden modificarse/eliminarse.
RES-05	El cierre de turno debe ocurrir dentro del horario de operación definido en configuración del sistema.

Regla de Negocio	Descripción
RN-CB-001	El <code>saldoInicial</code> debe ser $\geq 0$ y coincidir con el último <code>saldoFinal</code> .
RN-CB-002	El $\text{saldoFinal} = \text{saldoInicial} + \sum \text{ingresos} - \sum \text{egresos}$ ; debe coincidir con efectivo físico dentro de tolerancia.
RN-CB-003	Sólo el responsable original puede cerrar su <code>TurnoCaja</code> , salvo delegación explícita de un supervisor.
RN-CB-004	Diferencias mayores a la tolerancia generan alerta y bloquean el cierre hasta validación manual.

## 8. Valor Estratégico

ControlCajaBC aporta valor al negocio al:

Beneficio	Impacto en la Organización
Trazabilidad Financiera	Relaciona cada comprobante y movimiento con usuario y estación, garantizando trazabilidad y reduciendo pérdidas.

Eficiencia Operativa	Agiliza en un 50 % la apertura/cierre de turnos y reduce errores manuales.
Transparencia y Confianza	Provee datos en tiempo real y reportes confiables que fortalecen la confianza de dueños, contadores y auditores.
Escalabilidad Técnica	Permite escalar estaciones de caja sin comprometer el control, gracias a invarianza de TurnoCaja por estación.
Mejora Continua	Las métricas alimentan <b>IndicadoresNegocioBC</b> para optimizar políticas y procesos de caja.

# Aggregates, Entities, VO, Domains Events, Policies & Specifications\_cc

## 1. Entidades [🔗](#)

**Descripción general:** Una **Entidad** es un objeto con identidad propia cuya identidad persiste más allá de sus atributos. Se usa para representar conceptos del dominio que requieren rastrear su ciclo de vida.

### 1.1 TurnoCaja (Agregado Raíz) [🔗](#)

- Representa la sesión de caja de un usuario en un punto de venta.

- **Atributos:**

- `turnoId` (UUID).
- `responsableId` (UUID del usuario cajero).
- `codigoCaja` (identificador de la caja física, p. ej. "CAJA-01").
- `fechaHoraApertura` (DateTime).
- `saldoInicial` (VO Monto).
- `fechaHoraCierre` (DateTime, nulo mientras el turno está abierto).
- `saldoFinal` (VO Monto, calculado).
- `efectivoContado` (VO Monto, introducido al cierre).
- `diferenciaCaja` (VO Monto calculado).
- `estadoTurno` (enum ABIERTO/CERRADO).

- **Invariantes:**

- Solo puede haber **un TurnoCaja abierto por código de caja** a la vez (restricción de unicidad).
- No se puede **registrar ningún MovimientoCaja si el turno está cerrado** (un turno cerrado es inmutable en cuanto a movimientos).
- Un **turno abierto no puede cerrarse** sin antes haber registrado su conteo de efectivo (`efectivoContado`) y sin cumplir validaciones (p. ej., si hay diferencias excesivas, se requiere autorización adicional).
- Los **saldos deben cuadrar**: al cierre, `saldoFinal = saldoInicial + Σingresos - Σegresos` y la diferencia con `efectivoContado` debe estar dentro de la tolerancia permitida (reglas RN-CB-002 y RN-CB-004, ver sección 5).

- **Lifecycle:** Inicia con `estadoTurno = ABIERTO` tras ejecutar apertura. Durante la vida abierta, se agregan Movimientos. Al cierre, se fija `fechaHoraCierre`, se calcula `saldoFinal` y `diferenciaCaja`, y se marca `estadoTurno = CERRADO`. Después de cerrado, no se permiten más modificaciones; cualquier ingreso tardío (por ejemplo, un comprobante emitido después del cierre) debería registrarse en un nuevo turno (o reabrir uno nuevo).

Atributo	Descripción
<code>turnoId</code>	UUID único del turno

ResponsableId	Identificador del cajero asignado
fechaHoraApertura	Fecha y hora de inicio del turno
saldoInicial	Monto de efectivo al abrir caja
fechaHoraCierre	Fecha y hora de cierre del turno
saldoFinal	Monto final esperado antes de conteo físico
efectivoContado	Monto físico reportado en arqueo
diferenciaCaja	Diferencia entre saldoFinal y efectivoContado
estadoTurno	ABIERTO o CERRADO
codigoCaja	Identificador de la caja física

- **Comportamiento:**

- Apertura y cierre de turno
- Cálculo y conciliación de saldos
- Emisión de eventos de dominio ( TurnoCajaAbierto , TurnoCajaCerrado )

- **Relaciones:** Posee múltiples MovimientoCaja y un ResponsableCaja .

### 1.2 MovimientoCaja (Entidad dentro de TurnoCaja) ↗

- **Atributos:** movimientoId (UUID), tipoMovimiento (INGRESO/EGRESO), origenMovimiento (EVENTO\_COMPROMONTE, MANUAL, AJUSTE, etc.), monto (VO Monto > 0), descripcion (texto opcional explicativo, obligatorio para egresos manuales, por ejemplo), fechaHoraMovimiento (VO FechaHora) y posibles referencias a comprobanteId o clienteId según el caso.
- **Relación:** pertenece a un TurnoCaja (posee clave foránea turnoId). No tiene vida fuera del agregado; siempre es gestionado a través de TurnoCaja.
- **Invariantes/Reglas locales:**
  - El monto debe ser positivo (no se permiten movimientos de valor cero o negativo).
  - Si origenMovimiento == EVENTO\_COMPROMONTE , entonces debe existir un comprobanteId válido y asociado (garantizando integridad con ventas registradas). Además, movimientos automáticos importados de comprobantes no deben ser editables o eliminables manualmente (el sistema podría prohibir su anulación a menos que venga del evento opuesto, ComprobanteAnulado).
  - Si es un **movimiento de anulación** (reverso), su tipo será opuesto al original (un ingreso anulado se reflejará como egreso por igual monto, y viceversa).
- **Comportamiento:** Al crearse via TurnoCaja, ajusta el saldo actual del turno: suma el monto si es ingreso, resta si es egreso. Puede desencadenar validaciones en TurnoCaja (por ejemplo, RN-CB-002 recalcular saldo final continuamente). La anulación de un movimiento se implementa creando otro movimiento inverso (esto conserva trazabilidad en lugar de simplemente borrar el movimiento original).

Atributo	Descripción
movimientoId	UUID único del movimiento
tipoMovimiento	Enumeración ( INGRESO , EGRESO )
origenMovimiento	Enumeración ( EVENTO_COMPROBANTE , MANUAL )
monto	Monto del movimiento (>0)
descripcion	Motivo o descripción breve
fechaMovimiento	Fecha y hora de registro
clienteId	(Opcional) Identificador del cliente relacionado
comprobanteId	(Opcional) Identificador del comprobante asociado

- **Comportamiento:** Validar monto, aplicar tipo, actualizar saldo raíz.
- **Relaciones:** Pertenece a un TurnoCaja .

### 1.3 ResponsableCaja

- **Atributos:** `responsableId` (UUID del usuario que abrió el turno), `nombreUsuario` o datos básicos (opcional, para logging), `codigoCaja` (identificador de la caja física usada, mismo valor que en TurnoCaja), `rolUsuario` (rol o categoría del usuario, ej. Cajero, Supervisor) y tal vez un indicador de tenencia/empresa si aplica multi-empresa (p. ej., `tenantId` para identificar la empresa en contextos multi-tenant).
- **Relación:** Un TurnoCaja tiene exactamente un ResponsableCaja asociado (composición o referencia interna).
- **Responsabilidades:** Encapsula la identificación del actor que opera la caja. Se utiliza para:
  - Verificar **permisos**: por ejemplo, al abrir el turno se verifica que el usuario tenga permiso “AbrirCaja”, al cerrar que tenga “CerrarCaja” (o “CerrarCajaDelegada” si no es el mismo usuario). Estas comprobaciones pueden implementarse aquí o en un servicio de dominio, pero ResponsableCaja provee los datos (rol/permisos) necesarios.
  - Asociar la operación de caja con un **usuario específico** y **punto de venta**. Esto queda plasmado en eventos de dominio (por ejemplo, `TurnoCajaAbierto` lleva `responsableId` y `codigoCaja` para que otros contextos sepan quién y dónde se abrió la caja).
- **Comportamiento:** típicamente, esta entidad no “actúa” por sí sola sino a través de TurnoCaja. Pero podría exponer métodos de conveniencia como `esMismoUsuario(usuarioId)` o `tieneRol(rolRequerido)` para ayudar en lógica de negocio dentro del agregado (por ejemplo, RN-CB-003 que exige que solo el responsable original pueda cerrar su turno salvo delegación).

### 1.4 Otras Entidades (Externas al agregado principal)

ControlCajaBC es relativamente acotado en cuanto a entidades propias; la mayor parte del modelo son el TurnoCaja y sus componentes. No obstante, interactúa con **entidades de otros bounded contexts** a través de identificadores o datos

replicados. Por ejemplo, puede almacenar localmente un registro mínimo del cliente si se hacen ventas a crédito (para mostrar nombre del cliente en un ingreso “a cuenta”) o referencias a comprobantes electrónicos (IDs) para conciliación. Estas no son entidades plenas dentro de ControlCajaBC, sino **referencias** (IDs u objetos de valor representando entidades externas). Por ejemplo:

- *Cliente (referencia)*: El `clienteId` asociado a un movimiento de tipo ingreso a crédito (venta fiada) permite luego a GestiónClientesBC asociar ese pago. ControlCajaBC no gestiona la entidad Cliente, solo guarda su ID y quizás el nombre en el momento de la transacción para fines de recibo.
- *Comprobante (referencia)*: Similarmente, `comprobanteId` almacenado en MovimientoCaja es la referencia a la entidad ComprobanteElectrónico emitida en otro contexto. Esto mantiene la trazabilidad sin que ControlCajaBC necesite conocer internamente la estructura del comprobante.

No se definen nuevas entidades agregadas en ControlCajaBC más allá de las mencionadas, ya que su función es sobre todo registrar transacciones de efectivo en torno a un Turno.

## 2. Objetos de Valor (Value Objects) ☈

Además de los VO descritos, se pueden resaltar sus **invariantes y usos específicos** en el modelo:

- **CódigoCaja**: Debe ser único entre turnos abiertos. Una posible invariante es que no puede ser nulo ni vacío, y podría seguir un formato establecido (ejemplo: prefijo “CAJA-” seguido de número). ControlCajaBC podría obtener los códigos válidos desde ConfiguraciónSistemaBC (si las cajas están pre-configuradas) o simplemente usarlos como identificadores libres provistos en la petición de apertura. Este VO se usa al abrir turno para bloquear aperturas duplicadas (ver RES-01).
- **Monto**: Además de exigir valor no negativo, respeta la precisión configurada (por ejemplo, 2 decimales para soles). También puede contener la información de moneda (e.g., “PEN” o “USD”) si Factura Fácil soporta multi-moneda – aunque es probable que el sistema opere en una moneda base definida en ConfiguraciónSistemaBC. Todas las operaciones aritméticas de dinero en ControlCajaBC se realizan usando este VO, lo que centraliza el manejo de redondeos según la tolerancia configurada (para evitar diferencias de centavos).
- **FechaHora**: Lleva la zona horaria del tenant o empresa. En una arquitectura SaaS multi-tenant, distintos negocios podrían operar en diferentes zonas horarias, por lo que este VO asegura que las marcas de tiempo (especialmente de apertura/cierre) queden correctamente registradas con su contexto. Invariante: siempre debe tener zona horaria; no se acepta naive timestamps.
- **TipoMovimiento**: Sólo tiene dos valores posibles {INGRESO, EGRESO}. Invariante: obligatorio asignarlo en cada MovimientoCaja; determina la lógica de suma/resta en los cálculos. También podría ser usado para filtrar movimientos (ej. para sumatorias de solo ingresos o solo egresos en reportes).
- **SaldoCaja (VO compuesto)**: No se almacena directamente como atributo en TurnoCaja (ya que sus componentes se derivan), pero conceptualmente se puede construir al vuelo. Sus invariantes vienen de las relaciones matemáticas: `saldoActual = saldoInicial + ingresosTotales - egresosTotales`, `diferenciaCaja = saldoActual - efectivoContado`. Una invariante de negocio es que, **al momento de cierre**, `saldoFinal` debería igualar `saldoActual` y después de ingresar `efectivoContado`, la diferencia se calcula. Durante un turno abierto, el “saldoActual” va cambiando con cada movimiento; al cerrar, se fija ese como saldo final. Si se registra un ajuste de saldo por discrepancia, podría considerarse que se modifica el `saldoFinal` post-cierre para cuadrar con el efectivo (pero manteniendo registro del ajuste como movimiento). En general, **SaldoCaja** encapsula la regla de cálculo RN-CB-002 (ver sección 5) para verificar que los números cuadren dentro de tolerancia

### 3. Servicios de Dominio (interfaces y contratos)

Como mencionado, los Domain Services en ControlCajaBC definen comportamientos que pueden involucrar lógica compleja o coordinación con otros contextos. A continuación se perfilan con más detalle algunos métodos esperados y sus contratos:

- **ConciliacionService** (servicio de conciliación de caja con ventas):

- `registrarVentaEnCaja(comprobanteData)` – Método llamado cuando se emite un comprobante de venta. **Parámetros**: datos mínimos de la venta, p. ej. `{ comprobanteId, montoTotal, tipoPago, fechaHora }`. **Comportamiento**: Busca el TurnoCaja abierto correspondiente (por caja y quizás por usuario), luego invoca `TurnoCaja.registrarMovimiento(tipo=INGRESO, origen=EVENTO_COMPROBANTE, monto=montoTotal, ref=comprobanteId)`. Si no hay turno abierto en esa caja, podría lanzar excepción o (política a definir) crear un “turno por omisión” para que no se pierda el movimiento. **Resultado**: normalmente void (pero podría retornar el `movimientoId` generado). Este servicio actúa de consumidor del evento `ComprobanteEmitido` externo, adaptándolo al comando interno de registrar movimiento.
- `revertirVentaEnCaja(comprobanteData)` – **Parámetros**: `{ comprobanteId, motivoAnulacion }`. **Comportamiento**: Similar al anterior, ubica el TurnoCaja que contenga un movimiento asociado a ese comprobante y aún esté abierto, luego crea un movimiento inverso: `TurnoCaja.registrarMovimiento(tipo=EGRESO, origen=EVENTO_COMPROBANTE, monto=totalComprobante, descripcion="Anulación comp. ${comprobanteId}")`. Esto anula el ingreso. **Resultado**: void o `movimientoId` del egreso creado. Este método responde al evento `ComprobanteAnulado`.

- **PermisosService / AutorizacionService** (servicio de validación de permisos de usuarios para operaciones de caja):

- `validarApertura(usuario, codigoCaja)` – Verifica que el `usuario` tenga rol/permisos necesarios para abrir caja en esa estación. Podría internamente consultar un ACL (Anti-Corruption Layer) que comunica con Identidad&Autenticación o usar datos replicados de ConfiguracionSistemaBC. **Contrato**: devuelve true/false o lanza excepción de seguridad si no autorizado (por ejemplo, si el usuario no tiene permiso “Caja\_Abrir” o si ya existe un turno abierto en `codigoCaja` por otro usuario).
- `validarCierre(usuario, turno)` – Verifica que el `usuario` esté autorizado a cerrar el turno indicado. Normalmente, solo el mismo usuario que abrió (ResponsableCaja) puede cerrar (RN-CB-003), excepto que el usuario actual tenga rol Supervisor y un permiso especial para delegar cierre. **Contrato**: retorna true si puede cerrar, false o excepción si no (p. ej., “Permiso insuficiente para cerrar turno de otro usuario”). Este servicio podría usar datos de `ResponsableCaja` (rol original) y del usuario actual que solicita.
- `puedeAjustarSaldo(usuario)` – Verifica si el usuario (típicamente Supervisor) puede realizar un **ajuste de saldo** cuando hay diferencia fuera de tolerancia. Esto podría mapear a un permiso como “Caja\_Ajustar” definido en el sistema. Retorna boolean.

- **AjusteSaldoService** (servicio para manejo de discrepancias):

- `registrarAjuste(turno, nuevoEfectivoContado)` – Aplica cuando tras conteo inicial hay diferencia > tolerancia y un supervisor decide **ajustar** el monto contado (quizá tras re-conteo) antes de cerrar. **Parámetros**: referencia al `turno` abierto y el `nuevoEfectivoContado` confirmado.

**Comportamiento:** Recalcula la `diferenciaCaja` con el nuevo valor, y si ahora entra en tolerancia, permite proceder con el cierre. Podría registrar un evento o un log indicando que hubo un ajuste manual.

**Resultado:** actualiza el `TurnoCaja` (o devuelve un objeto de resultado con estatus de que ya se puede cerrar).

- `generarMovimientoAjuste(turno, tipo, monto, motivo)` – Alternativamente, si se quiere reflejar explícitamente el ajuste en la caja, este método crearía un `MovimientoCaja` de ajuste (tipo INGRESO si faltaba dinero – alguien lo repuso; o EGRESO si sobraba dinero – se retira el excedente).

**Contrato:** retorna el `movimientoId` de ajuste creado. Este movimiento se marcaría con un `origenMovimiento = AJUSTE` para diferenciarlo.

Cada servicio de dominio normalmente se implementa como una **interfaz** dentro del dominio (definiendo métodos) con una posible implementación en la capa de dominio o aplicación. Por ejemplo, `ConciliacionService` podría tener implementaciones distintas si la orquestación de eventos varia (p.ej., stub para pruebas unitarias vs implementación real). Los **contratos** arriba descritos (parámetros y resultados) se mantendrían constantes. Además, estos servicios idealmente exponen **interfaces claras** para que la capa de aplicación los use sin conocer detalles internos (por ejemplo, la app service llamará a `conciliacionService.registrarVentaEnCaja(datosComprobante)` al recibir un evento de venta). Esto sigue el principio DDD de **Infrastructure Ignorance**: el dominio define qué hacer; cómo se recibe/entrega (mensajería, REST) es manejado por adaptadores en otra capa.

## 4. Eventos de Dominio

`ControlCajaBC` publica varios **eventos de dominio internos** cuando ocurren situaciones relevantes en el modelo. Estos eventos describen **hechos pasados en el dominio de caja** que interesan a otros bounded contexts o a componentes externos. Los principales eventos internos definidos son:

- **TurnoCajaAbierto** – Indica que se ha abierto un nuevo turno de caja. **Origen:** se genera al completar exitosamente el caso de uso “Apertura de Turno de Caja”. **Payload:** contiene el ID del turno (`turnoId`), el ID del usuario responsable (`responsableId`), el código de la caja (`codigoCaja`), y la fecha/hora de apertura (`fechaHoraApertura`). No incluye el saldo inicial (pues es conocido por contexto interno, aunque podría añadirse si fuera útil). **Consumidores:** típicamente el contexto `IndicadoresNegocioBC` lo consume para iniciar métricas de flujo de caja (ej. contar un nuevo turno abierto en dashboards). También la interfaz de usuario (UI) podría escucharlo para habilitar opciones de registrar ventas/movimientos en esa caja. **Reglas asociadas:** asegura que RN-CB-001 (saldo inicial  $\geq 0$ ) fue validada antes de emitirlo.
- **MovimientoCajaRegistrado** – Notifica que un movimiento (ingreso/egreso) ha sido registrado en un turno. **Origen:** caso de uso “Registro de Movimiento de Caja” (manual o automático). **Payload:** incluye `movimientoId`, `turnoId` al que pertenece, `tipoMovimiento` (INGRESO/EGRESO), `monto` del movimiento y la fecha/hora del mismo. (Podría extenderse con un campo que indique si es automático o manual, o el comprobanteId, pero en general esos detalles los tiene internamente el contexto). **Consumidores:** `IndicadoresNegocioBC` para actualizar dashboards de ingresos diarios, ticket promedio, etc.. Un módulo de Auditoría podría almacenarlo para auditoría financiera. **Reglas asociadas:** refleja que RN-CB-002 (recálculo de saldos) y RN-CB-004 (posible diferencia en caso de egreso no autorizado) se aplicaron en el proceso. Observación: Si un movimiento se anula, se emite otro `MovimientoCajaRegistrado` (con un nuevo ID) representando la reversión, manteniendo así que cada ajuste también es un evento (como nota, la *no emisión* de un evento para anulación sería un anti-pattern, se prefiere emitir un evento de “`MovimientoAnulado`” o reutilizar el mismo evento indicando naturaleza inversa).
- **TurnoCajaCerrado** – Indica que un turno de caja ha sido cerrado. **Origen:** caso de uso “Cierre de Turno de Caja” (ya sea por el cajero responsable u otro autorizado). **Payload:** contiene `turnoId`, el `responsableId` (cajero original del turno), el `saldoFinal` calculado, la `diferenciaCaja` registrada y la `fechaHoraCierre`. (El

saldoFinal + diferencia implican también cuánto fue el efectivo contado, que podría deducirse: efectivoContado = saldoFinal - diferenciaCaja). **Consumidores**: IndicadoresNegocioBC lo usa para reportes de conciliación y posiblemente generar alertas si `diferenciaCaja` ≠ 0. También un servicio de generación de reportes PDF podría suscribirse para inmediatamente producir el informe de cierre cuando ocurre este evento. **Reglas asociadas**: garantiza que RN-CB-004 (diferencias > tolerancia) fue evaluada; si la diferencia era muy alta probablemente este evento no se hubiera emitido sin intervención (el cierre se bloquea hasta ajuste). Observación: Al emitirse, este evento marca el fin del turno, por lo que el sistema puede tomar acciones como bloquear nuevas ventas en esa caja hasta la próxima apertura.

**Contratos de Mensaje (Eventos Internos)**: Cada evento se publica generalmente en el **Event Bus** con un esquema JSON similar a:

- *Ejemplo JSON TurnoCajaAbierto :*

```
1 {
2   "codigoCaja": "CAJA-01",
3   "event": "TurnoCajaAbierto",
4   "fechaHoraApertura": "2025-07-04T17:10:00Z",
5   "responsableId": "user-12345",
6   "turnoId": "550e8400-e29b-41d4-a716-446655440000"
7 }
```

- *Ejemplo JSON MovimientoCajaRegistrado :*

```
1 {
2   "event": "MovimientoCajaRegistrado",
3   "fechaHoraMovimiento": "2025-07-04T18:30:25Z",
4   "movimientoId": "789e1200-e29b-41d4-a716-446655440000",
5   "monto": 350.50,
6   "tipoMovimiento": "INGRESO",
7   "turnoId": "550e8400-e29b-41d4-a716-446655440000"
8 }
```

- *Ejemplo JSON TurnoCajaCerrado :*

```
1 {
2   "diferenciaCaja": -20.00,
3   "event": "TurnoCajaCerrado",
4   "fechaHoraCierre": "2025-07-04T22:00:00Z",
5   "responsableId": "user-12345",
6   "saldoFinal": 1250.00,
7   "turnoId": "550e8400-e29b-41d4-a716-446655440000"
8 }
```

Estos contratos muestran los campos clave que viajan en cada evento. Nótese que el evento de cierre comunica el resultado financiero final (`saldoFinal` y `diferenciaCaja`), útil para que otros contextos (indicadores, alertas) reaccionen.

Además de los eventos internos, ControlCajaBC **consume eventos de otros contextos (eventos inbound)**, aunque esos no son definidos por ControlCajaBC pero sí relevantes en su modelo. Los más importantes recibidos son:

- `ComprobanteEmitido` (from ComprobantesElectonicosBC) – Provoca un ingreso automático (ya detallado). Su mensaje contiene al menos `comprobanteId`, `montoTotal`, `tipoPago`, `fechaHora`.
- `ComprobanteAnulado` (from ComprobantesElectonicosBC) – Provoca una reversión (egreso). Contiene `comprobanteId` y `motivoAnulacion` (y se podría inferir el monto a partir del ID buscando internamente, o

idealmente se pasaría también el `monto` ).

- `ClienteActualizado` (from `GestionClientesBC`) – Informa cambios en estado o límite de crédito de un cliente. ControlCajaBC podría usar estos datos para validar al registrar un ingreso a cuenta de cliente (por ejemplo, no permitir si el cliente excedió su límite de crédito). El payload incluye `clienteId`, nuevo `estadoCliente`, nuevo `limiteCredito`, etc.
- `PermisosUsuarioActualizado` (from `ConfiguracionSistemaBC`) – Notifica cambios de roles/permisos de un usuario. Payload: `usuarioId`, lista de permisos/roles vigente. ControlCajaBC lo usa para actualizar la capacidad de ese usuario de abrir/cerrar caja. Por ejemplo, si a un cajero se le revoca el permiso de cierre mientras tiene un turno abierto, podría requerir que un supervisor lo cierre.

Estos eventos inbound actúan como **gatillos** de casos de uso en ControlCajaBC (ver sección 3). Por ejemplo, `ComprobanteEmitido` dispara el flujo “Registro de Movimiento de Caja (automático)”, y `PermisosUsuarioActualizado` puede afectar las políticas de seguridad en curso (aunque típicamente la aplicación ya consultaría permisos actualizados en cada acción).

#### 4. Políticas (Policies)

**Descripción general:** Reglas de alto nivel que controlan validaciones transversales.

Código	Regla / Descripción
RN-CB-001	<code>saldoInicial</code> debe ser $\geq 0$ y coincidir con último <code>saldoFinal</code> .
RN-CB-002	$\text{saldoFinal} = \text{saldoInicial} + \sum \text{ingresos} - \sum \text{egresos}$ ; validar tolerancia.
RN-CB-003	Sólo el responsable puede cerrar turno, salvo delegación explícita.
RN-CB-004	Diferencias $>$ tolerancia generan alerta y bloquean cierre hasta validación manual.

**Control de reglas:**

Rango códigos	Tema	Total	Próx. RN	Observaciones
RN-CB-001–RN-CB-004	ControlCajaBC	4	RN-CB-005	Definir RN para tolerancia dinámica

#### 5. Especificaciones (Specifications)

**Descripción general:** Specs encapsulan criterios de negocio reutilizables.

Caso	Precondiciones	Reglas Aplicadas
RegistrarMovimientoValido	TurnoCaja ABIERTO	RN-CB-002

CerrarTurnoCaja	TurnoCaja ABIERTO , sin movimientos pendientes	RN-CB-001, RN-CB-004
-----------------	--	----------------------

- **Propósito:** Garantizar validaciones antes de ejecutar acciones críticas.

## 6. Políticas de Dominio y Especificaciones (Detallado) ↗

En ControlCajaBC, existen **políticas de negocio** que transversalmente afectan las validaciones y comportamientos, así como **especificaciones** que encapsulan ciertas reglas para reutilizarlas. Algunas destacadas:

- **Política de Tolerancia de Diferencia:** Define cuánto **descuadre** se permite al cerrar un turno sin requerir intervención. Puede ser un monto fijo (ej. S/5.00) o un porcentaje del total manejado. Esta política se aplica durante el cierre: si `|diferenciaCaja| > tolerancia`, entonces **bloquea el cierre** (no se emite `TurnoCajaCerrado`) y requiere acción de un supervisor (RN-CB-004). La tolerancia en sí puede provenir de ConfiguracionSistemaBC como parámetro configurable.
- **Política de Horario de Operación:** Restringe que los cierres de caja ocurran dentro de cierto horario (por ejemplo, al final del día laboral). Si un usuario intenta cerrar fuera del rango (por ejemplo, después de medianoche cuando el negocio está cerrado), podría impedirse o marcarse para auditoría (RES-05: “El cierre de turno debe ocurrir dentro del horario de operación definido”). Esto evita prácticas indebidas como cerrar caja a deshoras.
- **Política de Unicidad de Turno:** Ya mencionada, implementa RES-01: “Solo un TurnoCaja abierto por estación simultáneamente”. Esto se aplica como una verificación global antes de abrir un nuevo turno (posiblemente una consulta al repositorio para ver si hay algún turno abierto con ese CódigoCaja). Se podría implementar vía una **Specification** `TurnoAbiertoSpec(codigoCaja)` que consulta repositorio; si devuelve true (existe un turno abierto en esa caja), la apertura es rechazada.
- **Política de Permisos por Rol:** Asocia operaciones a roles: por ejemplo, `AbrirCaja` solo Cajero o Supervisor; `CerrarCaja` Cajero (propio turno) o Supervisor (delegado); `AnularMovimiento` solo Supervisor si es automático; `AjustarSaldo` solo Supervisor. Estas reglas se mapearon a RN-CB-003 (solo responsable cierra, salvo delegación), y complementariamente a permisos (“`CerrarCajaDelegada`”) para el supervisor. Se implementan chequeando roles del usuario en cada caso de uso (ver sección 12 Seguridad). En el dominio, esto podría ser una **especificación** `UsuarioAutorizadoSpec(operacion)` que verifica si el usuario actual cumple los roles requeridos para la operación dada.
- **Specification de MontoPositivo:** valida que cualquier valor monetario ingresado (saldo inicial, montos de movimientos, efectivo contado) sea  $\geq 0$ . Se utiliza en varios lugares (apertura, registrar movimiento, cierre). Implementada una vez, puede reutilizarse en métodos de agregado para lanzar errores de validación uniformes.
- **Specification de SaldoCuadrado:** podría encapsular la comparación entre efectivo contado y saldo esperado dentro de tolerancia. En la etapa de cierre, en lugar de espacer la lógica de tolerancia, una Specification `SaldoDentroDeTolerancia` toma `diferenciaCaja` y devuelve true si `|diferencia| <= toleranciaConfigurada`. Esto se usa para decidir si permitir el cierre directo o activar la política de ajuste/alerta.
- **Specification TurnoSinMovimientosPendientes:** aplicable antes de cerrar, verifica que no haya movimientos “pendientes” (por ejemplo, comprobantes emitidos en proceso o movimientos temporales). En esta implementación, podríamos interpretarlo como “no hay movimientos en estado inconsistente”. Si los movimientos se registran atómicamente, esta spec siempre pasaría; pero si hubiera un concepto de movimientos no confirmados, se chequearían aquí.

**Ejemplo de uso de Especificaciones (en pseudocódigo):** Antes de cerrar un turno:

```

1 if (!SaldoDentroDeToleranciaSpec.isSatisfiedBy(turno)) {
2     throw new BusinessException("Diferencia excede tolerancia, requiere ajuste");
3 }
4 if (!TurnoSinMovimientosPendientesSpec.isSatisfiedBy(turno)) {
5     throw new BusinessException("Existen movimientos pendientes por procesar");
6 }

```

Estas especificaciones encapsulan esas reglas de manera reutilizable y separable, pudiendo ser probadas aisladas y combinadas para políticas más complejas.

## 2.2 Políticas de Negocio y Ejemplos de Mensajes

Además de las políticas mencionadas, enumeramos algunas **reglas de negocio concretas (RN)** y **restricciones (RES)** definidas en este dominio, con su código identificador, para garantizar la comprensión y cumplimiento:

- **RES-01:** Solo puede existir **un turno abierto por cada estación de caja** a la vez. *Detalle:* Si ya hay un TurnoCaja con `estado = ABIERTO` para cierto `codigoCaja`, cualquier intento de abrir otro es bloqueado.
- **RES-02:** No se pueden registrar **movimientos** si no hay un turno abierto (es decir, si la caja está cerrada). *Detalle:* La aplicación debe impedir operaciones de ingreso/egreso a menos que `TurnoCaja.estadoTurno == ABIERTO`.
- **RES-03:** Los movimientos automáticos (derivados de eventos de venta) requieren un **comprobanteId válido** asociado. *Detalle:* Esto asegura que cada ingreso automático realmente corresponda a una venta real; de lo contrario se rechaza.
- **RES-04:** Los movimientos automáticos son **inmutables**: no pueden ser modificados ni eliminados manualmente. *Detalle:* Si se necesita corregir uno, debe venir una anulación del contexto de comprobantes; no se permite a un usuario editar montos de ventas registradas en caja.
- **RES-05:** El **cierre de turno** debe efectuarse dentro del **horario de operación** definido en la configuración del sistema. *Detalle:* Por ejemplo, si la empresa define que la caja debe cerrarse antes de las 8pm, intentar cerrarla después podría requerir rol especial o dejar registro de excepción.

Y las principales **Reglas de Negocio (RN)** que rigen cálculos y procesos internos:

- **RN-CB-001:** El `saldoInicial` al abrir turno **debe ser ≥ 0** y debe coincidir con el último `saldoFinal` del turno anterior en esa misma caja (asumiendo que se arrastra el efectivo). *Ejemplo:* Si ayer se cerró la caja con S/1000, hoy al abrir en la mañana `saldoInicial` debería declararse S/1000 (lo que quedó en caja).
- **RN-CB-002:** El `saldoFinal` de un turno se calcula como `saldoInicial + Σ(ingresos) - Σ(egresos)`, y este valor **debe coincidir con el efectivo físico (efectivoContado)** dentro de un margen de tolerancia aceptable. *Detalle:* Si la diferencia excede la tolerancia (ver RN-CB-004), se considera incumplida esta regla. Esta es la fórmula fundamental de cuadratura de caja.
- **RN-CB-003:** Solo el **responsable original** del turno puede cerrarlo, **salvo** que exista una delegación explícita a otro usuario (e.g. un Supervisor). *Detalle:* Implementado vía permisos: un usuario distinto necesita permiso "CerrarCajaDelegada". Esta regla asegura responsabilidad personal en el manejo de caja.
- **RN-CB-004:** Si existen diferencias de caja mayores a la tolerancia configurada, el sistema debe **generar una alerta y bloquear el cierre** hasta que haya una validación o ajuste manual. *Detalle:* Esto implica que ante una gran discrepancia, el evento `TurnoCajaCerrado` no se emite y quizás se emite en su lugar un evento de alerta (o se notifica al usuario que requiere verificación).
- **(Propuesta) RN-CB-005:** (Regla planeada) Definir manejo de tolerancias dinámicas según monto manejado. *Detalle:* Por ejemplo, tolerancia de S/1 para cajas con poco movimiento, pero hasta S/10 para cajas con volumen alto. Aún no implementada completamente (indicada como próxima RN a definir).

Estas reglas y restricciones garantizan la integridad del proceso de caja. A continuación, se mencionan algunos ejemplos de mensajes de error o JSON de comandos que ilustran cómo se aplican:

- Si un cajero intenta **abrir caja** y ya hay un turno abierto en esa estación, el sistema respondería con un error JSON:

```
1 {
2   "error": "TurnoPendiente",
3   "message": "Ya existe un turno abierto en CAJA-01. Debe cerrarlo antes de abrir uno nuevo."
4 }
```

(Aplica RES-01).

- Al intentar **registrar un movimiento** sin turno abierto:

```
1 {
2   "error": "OperacionInvalida",
3   "message": "No hay un turno de caja abierto. Por favor abra caja antes
4   de registrar movimientos."
5 }
```

(Aplica RES-02).

- Al **cerrar turno** con una diferencia fuera de tolerancia:

```
1 {
2   "error": "DiferenciaExcesiva",
3   "message": "Existe una diferencia de S/20.00 en la caja,
4   superior a la tolerada. Cierre bloqueado hasta ajuste por supervisor."
5 }
6 }
```

(Aplica RN-CB-004: *bloquea cierre y requiere intervención*).

Estos mensajes evidencian cómo las reglas de negocio se comunican al usuario o al sistema que interactúa. Las **políticas** se configuran en un nivel y se aplican consistentemente a través de *Specifications* o validaciones en los casos de uso, asegurando un comportamiento uniforme del contexto de ControlCajaBC. En el siguiente apartado, veremos los casos de uso en detalle, lo que permitirá entender cómo estas reglas se entrelazan en los flujos.

## Casos de uso\_cc

### 1. ¿Qué son los Casos de Uso en DDD? ☰

Los **Casos de Uso** describen cómo los **actores** (usuarios o sistemas externos) interactúan con el dominio para cumplir objetivos de negocio.

A continuación, se describen detalladamente los principales **Casos de Uso** de ControlCajaBC, incluyendo sus actores, pasos (flujos principal y alternos), los eventos y entidades involucrados, así como diagramas de secuencia para ilustrar la interacción entre el actor (usuario u otro sistema) y el Bounded Context. Cada caso de uso refleja el **Lenguaje Ubícuo** y las reglas de negocio previamente definidas.

En DDD, los Casos de Uso sirven para:

- Definir los **servicios de aplicación** que orquestan agregados y entidades.
- Reflejar el **Ubiquitous Language**.
- Guiar el diseño de flujos, validaciones y eventos.

### 2. Resumen de Casos de Uso ☰

Nº	Caso de Uso	Actor Principal
1	Apertura de Turno de Caja	Cajero
2	Registro de Movimiento de Caja	Cajero
3	Cierre de Turno de Caja	Cajero / Supervisor
4	Consulta de Balance Actual	Cajero / Gerente
5	Consulta de Historial de Movimientos	Cajero / Auditor
6	Descarga de Reporte de Cierre	Cajero / Gerente
7	Anulación de Movimiento de Caja	Cajero / Supervisor
8	Ajuste de Saldo por Discrepancia	Supervisor
9	Delegar Cierre de Turno	Supervisor

## 2.1 Apertura de Turno de Caja

**Actor Principal:** Cajero (usuario que inicia la caja).

**Descripción:** El cajero inicia un nuevo turno de caja indicando el saldo inicial en efectivo. Esto crea un TurnoCaja en estado abierto para comenzar a registrar operaciones.

**Precondiciones:**

- El usuario cajero debe tener permiso de *AbrirCaja* (estar autorizado a manejar caja).
- No debe existir ya un turno abierto en la misma estación/caja (misma *codigoCaja*) – es decir, debe cumplirse RES-01.

**Flujo Principal:**

1. *Input:* El Cajero indica el **saldoInicial** con el que cuenta al comenzar (dinero físico en la caja registradora).
2. El sistema (**ControlCajaBC**) valida que el usuario esté autorizado (permiso *AbrirCaja*) y verifica que **no haya un turno activo** actualmente en esa caja. Si alguna validación falla, aborta (ver flujos alternos).
3. Se crea un nuevo **TurnoCaja** con estado = ABIERTO, registrando: responsable (usuario), estación (*codigoCaja*), hora de apertura actual, saldoInicial proporcionado, y saldos acumulados iniciados en cero.
4. El sistema **publica el evento** de dominio **TurnoCajaAbierto**, conteniendo los datos del turno abierto (*turnoId*, *responsableId*, etc.).
5. *Output:* Se devuelve la confirmación al cajero con los datos del turno abierto (p. ej., código/ID de turno, hora de apertura, saldoInicial). La UI podría mostrar un mensaje “Turno de Caja abierto exitosamente” junto con el saldo inicial.

**Flujos Alternativos:**

- **A1: Usuario sin permiso** – Si en el paso 2 el cajero no tiene los privilegios necesarios, el sistema responde con error “No autorizado”. *Resultado:* el turno no se abre.
- **A2: Turno ya activo** – Si ya existe un turno abierto (detected en paso 2), se rechaza la apertura y se notifica al usuario con un mensaje del tipo “Ya existe un turno pendiente abierto”, indicando que debe cerrarlo primero.
- (Si hubiera otros errores, como saldoInicial negativo, se manejarían aquí: se rechaza y se informa el motivo, ej. “Saldo inicial inválido”. Esta validación corresponde a RN-CB-001 y evitaría incluso llegar a crear el turno).

**Eventos Generados:** **TurnoCajaAbierto** (publicado solo si el flujo principal llega hasta el paso 4).

**Entidades Afectadas:** Crea una instancia de **TurnoCaja** (y su ResponsableCaja asociada). No hay movimientos en este caso aún.

**Diagrama de Secuencia (Apertura de Turno):**



## 2.2 Registro de Movimiento de Caja

**Actor Principal:** Cajero.

**Descripción:** Agrega un nuevo movimiento de dinero en el turno abierto actual, ya sea un ingreso (p.ej. venta, ingreso de efectivo) o un egreso (p.ej. gasto, retiro). Puede ser invocado manualmente por el cajero (ej.: registrar un gasto menor) o automáticamente por el sistema al generarse una venta pagada en efectivo.

**Precondiciones:** Debe existir un **TurnoCaja abierto** (*TurnoCaja.estado* = ABIERTO) en la caja actual. (Además, si es un movimiento manual, el usuario debe tener permiso para registrar movimientos; normalmente cualquier cajero con turno abierto puede hacerlo).

### Flujo Principal (Movimiento manual o general):

1. *Input:* El Cajero (o el sistema) envía los datos del movimiento a registrar: el **tipoMovimiento** (INGRESO o EGRESO), el **monto** correspondiente, y adicionalmente o bien un **comprobanteId** si es asociado a una venta o un campo de **descripción** si es un movimiento manual no vinculado a comprobante.
2. ControlCajaBC verifica:
  - Que el **monto** sea > 0 (regla de validación básica).
  - Calcula preliminarmente el nuevo saldo (saldoActual + monto o - monto) y evalúa RN-CB-002: que el saldo final se mantenga consistente; en este paso puede no haber problema ya que solo se suma/resta.
3. El sistema crea un **MovimientoCaja** dentro del TurnoCaja abierto, con los datos proporcionados. Actualiza en el agregado TurnoCaja su **saldoActual** sumando (si ingreso) o restando (si egreso) el monto.
4. Tras agregar el movimiento, el TurnoCaja recalcula sus totales (**ingresosTotales** o **egresosTotales**) y por tanto actualiza internamente su **saldoActual**. (No se cierra aún, solo queda listo con nuevo saldo).
5. El sistema **publica el evento** **MovimientoCajaRegistrado**, con los detalles del movimiento (ID, turnoId, tipo, monto, fecha).
6. *Output:* Se confirma al cajero la operación registrada (p.ej., mostrando en pantalla el nuevo saldo actual de caja, o listando el movimiento recién añadido). En un flujo automático (como venta), esta confirmación puede ser implícita, integrándose en la confirmación de venta.

### Flujos Alternativos:

- **B1: Monto inválido** – Si el **monto** enviado es 0 o negativo, el sistema rechaza el movimiento con un error de validación (ej. “El monto debe ser mayor a 0”). No se crea movimiento.
- **B2: RN-CB-002 incumplida** – Esta regla básicamente asegura consistencia de saldos; durante la ejecución de un movimiento en sí no suele fallar salvo por limitaciones como precision/tolerancia. Un caso podría ser: si sumando ese movimiento se excede algún valor máximo permitido para la caja (no se definió un tope, pero podría existir). O, en un contexto extendido, si es un **ingreso a crédito** y el cliente excede su límite de crédito, podría considerarse dentro de este flujo: se bloquearía con alerta “El cliente excede su límite de crédito, no se puede registrar el cobro” – aunque esto involucra interacción con GestiónClientesBC. Para mantenerlo aquí: supongamos que RN-CB-002 podría fallar por un error interno de cálculo; el sistema entonces haría rollback y presentaría una alerta. (En la práctica, RN-CB-002 es chequeo de fórmula, así que no falla en un solo movimiento; la validación real de RN-CB-002 ocurre al cierre).
- **B3: Turno no abierto** – Si no hay turno abierto, este flujo realmente no debería iniciarse (está cubierto en precondition), pero si ocurriera, se responde con error “No existe un turno abierto” (similar al error de RES-02).

**Eventos Generados:** **MovimientoCajaRegistrado** (si el movimiento se agregó correctamente).

**Entidades Afectadas:** Crea una instancia de **MovimientoCaja** (asociada al TurnoCaja vigente) y actualiza los atributos acumulados del **TurnoCaja** (saldoActual, etc.).

### Diagrama de Secuencia (Registro de Movimiento):



(En escenarios automáticos, el iniciador sería otro contexto enviando un evento, pero la secuencia dentro de ControlCajaBC es similar, solo que en lugar de Cajero habría, por ejemplo, ComprobantesBC enviando datos.)

## 2.3 Cierre de Turno de Caja

**Actor Principal:** Cajero (responsable del turno) o Supervisor (delegado).

**Descripción:** Concluye un turno de caja al final del periodo (ej. fin del día o fin de turno del cajero). Se realiza el conteo

físico del efectivo, se concilian los valores, se registra cualquier diferencia y se cierra formalmente el TurnoCaja con un estado final.

#### Precondiciones:

- Debe haber un **TurnoCaja abierto** para poder cerrarlo.
- No debe haber “movimientos en proceso” pendientes (p. ej., no estar en medio de registrar un movimiento). En este diseño asumimos que si se está invocando el cierre es porque ya terminaron de registrarse las ventas.
- Si el actor que solicita el cierre **no es el mismo** que abrió el turno, debe tener el permiso especial para cierre delegado (Rol Supervisor con permiso *CerrarCajaDelegada*). (RN-CB-003).
- Asegurar que todos los movimientos automáticos esperados llegaron (por ejemplo, no intentar cerrar mientras se está emitiendo una factura, aunque esto es difícil de saber; se deja a criterio del usuario y del negocio evitar cerrar en mitad de una transacción activa).

#### Flujo Principal:

1. *Input:* El Cajero (o Supervisor) indica que desea cerrar la caja, proporcionando el valor de **efectivoContado** – es decir, cuánto dinero en efectivo hay físicamente en la caja en ese momento. (Idealmente tras hacer el conteo de billetes/monedas).
2. El sistema calcula la **diferenciaCaja** = efectivoContado – saldoActual del turno. Luego valida la regla **RN-CB-004**: si la diferencia supera la tolerancia permitida, *no procede* con el cierre normal y en su lugar activa el flujo alternativo (ver A3). Si la diferencia está dentro de límites, continúa.
3. Se actualiza el TurnoCaja: se registra **fechaHoraCierre** (timestamp actual) y se calcula el **saldoFinal** = saldoActual (ya incluyendo todo movimiento hasta antes del cierre). Se marca **estadoTurno** = **CERRADO**. También se almacena el **efectivoContado** y la **diferenciaCaja** calculada. Básicamente, el turno pasa a estado cerrado con todos los datos de cierre.
4. El sistema **publica el evento** **TurnoCajaCerrado**, con el turnoId, responsableId, saldoFinal, diferenciaCaja y hora de cierre. Este evento notifica a otros contextos del cierre y de cualquier discrepancia.
5. *Post-acción:* El sistema genera el **reporte de cierre** (generalmente un PDF) consolidando la información del turno (saldo inicial, todos los ingresos/egresos, saldo final, efectivo contado, diferencia, firma del responsable, etc.). Este reporte puede guardarse y ofrecerse al usuario para descarga.
6. *Output:* Se confirma al usuario que la caja ha sido cerrada exitosamente. Si es el cajero mismo, probablemente se imprime o muestra el reporte de cierre. Si es un supervisor cerrando en delegación, igualmente se notifica el cierre (posiblemente al cajero original también, vía notificación).

#### Flujos Alternativos:

- **C1: Diferencia > tolerancia (descuadre)** – Si en el paso 2 la diferenciaCaja calculada excede la tolerancia definida (RN-CB-004), el sistema **no permite** el cierre automático. En su lugar:
  - Presenta una **alerta** indicando la discrepancia (por ejemplo: “Diferencia de S/20.00 excede lo permitido. Cierre bloqueado.”).
  - El turno podría quedar en un estado intermedio (no cerrado) hasta que se realice un ajuste. Aquí puede activarse un sub-flujo: **Ajuste de Saldo por Discrepancia (Caso 8)** donde un Supervisor interviene para corregir la situación (ver 3.8).
  - Solo tras resolver el ajuste (ej. actualizando efectivoContado o registrando un movimiento de ajuste) se reintenta el cierre (volviendo al paso 2 con diferencias ahora aceptables).
- **C2: Permiso insuficiente para cierre delegado** – Si quien invoca el cierre no es el responsable original y tampoco tiene permiso de Supervisor, el sistema lo rechaza de inmediato con error “No autorizado para cerrar turno de otro

usuario” (esto se checa en precondiciones). *Ejemplo:* un cajero intenta cerrar el turno de otro cajero sin ser supervisor. El sistema impediría siquiera ingresar efectivo contado en esa situación.

- **C3: Movimientos pendientes** – Si por alguna razón hay movimientos aún no confirmados (por ejemplo, en cola de un sistema externo), el sistema podría advertir “Existen transacciones en proceso, intente más tarde”. Este flujo es más aplicable en integraciones asíncronas; en este contexto, asumimos que no hay tales pendientes en el momento de cierre (o se maneja manualmente).

**Eventos Generados:** TurnoCajaCerrado (solo si se completó el cierre con éxito). Si el cierre fue bloqueado (flujo alternativo C1), en su lugar podríamos emitir una alerta de evento distinto o simplemente no emitir nada hasta resolver.

**Entidades Afectadas:** Actualiza el TurnoCaja (pasando a cerrado, con sus campos finalizados). Todos los MovimientoCaja quedan asociados a un turno cerrado (no cambian, pero desde este punto se consideran inmutables). Puede crear una entidad de ReporteCierre (como registro histórico, fuera del dominio principal pero relacionado).

#### Diagrama de Secuencia (Cierre de Turno):



## 2.4 Consulta de Balance Actual

**Actor Principal:** Cajero (que tiene el turno abierto) o Gerente.

**Descripción:** Permite obtener, en cualquier momento mientras la caja está abierta, un resumen del estado actual: el saldo acumulado hasta el momento y la diferencia preliminar si se hiciera un conteo (aunque usualmente diferencia solo se calcula al cierre con efectivo real). Básicamente muestra el saldoActual en tiempo real.

**Precondiciones:** Debe haber un turno abierto (TurnoCaja.estado = ABIERTO) para poder consultar su balance. Si la caja está cerrada o nunca abierta, no hay “balance actual” que consultar.

#### Flujo Principal:

1. *Input:* El actor solicita la operación de “Consultar Balance” (sin parámetros o especificando de qué turno, generalmente el turno abierto actual se asume).
2. El sistema localiza el TurnoCaja abierto del cajero (o de la estación).
3. Lee el saldoActual del agregado TurnoCaja. Este saldoActual es el cálculo actualizado tras los últimos movimientos registrados. También puede calcular la diferenciaCaja preliminar si el usuario ingresó un valor contado temporal (no es lo usual, normalmente diferencia = 0 durante el turno abierto ya que aún no se contó efectivo). En este caso, podríamos suponer que solo se devuelve el saldo actual y quizás el total de ingresos/egresos hasta el momento.
4. *Output:* El sistema devuelve la información requerida en formato JSON, por ejemplo: { saldoActual: 1250.00, diferencia: 0.00 }. Aquí diferencia podría ser siempre 0 durante el turno abierto, o no incluirse. Lo importante es el saldo. Esta respuesta se muestra al cajero o gerente en pantalla, posiblemente en un mini-reportaje o ventana.

#### Flujos Alternativos:

- **D1: Sin turno activo** – Si no se encuentra un turno abierto (por ejemplo, el cajero consulta fuera de turno o ya cerró), el sistema devuelve un error “Turno no encontrado o no abierto”. No hay datos que mostrar. El gerente también debe especificar cuál turno quiere ver: si consulta “balance actual” quizás se asume el turno activo de alguna caja; si ninguna, igual error.
- No hay otros alternos ya que es una consulta segura. Si hubiera múltiples turnos abiertos (lo cual no ocurre en la misma caja, pero un gerente podría tener varias cajas activas en distintos puntos de venta), tendría que seleccionar uno.

**Eventos Generados:** Ninguno (solo lectura, no altera estado).

**Entidades Afectadas:** Ninguna (solo se lee el TurnoCaja existente, no se modifica).

#### Diagrama de Secuencia (Consulta Balance):



### 2.5 Consulta de Historial de Movimientos

**Actor Principal:** Cajero (de su propio turno) o Auditor/Contador.

**Descripción:** Lista todos los movimientos registrados en un turno de caja, detallando ingresos y egresos, para revisión o auditoría. El objetivo es tener el desglose de transacciones que explican el saldo del turno.

**Precondiciones:** Debe existir el TurnoCaja cuyo historial se desea consultar. Puede ser un turno **ABIERTO o CERRADO** (se permite consultar historial de un turno ya cerrado también). El actor debe tener permiso para ver esa información (cajeros pueden ver sus propios movimientos; auditores o contadores pueden ver todos).

#### Flujo Principal:

1. *Input:* El actor proporciona una referencia del turno a consultar, típicamente `turnoId` (si el actor es auditor, debe especificar cuál turno; si es el cajero y su turno sigue abierto, el sistema podría asumir ese turno).
2. El sistema valida que el turno con ese `turnoId` exista (en el repositorio de TurnoCaja) y opcionalmente que el actor tenga derecho a verlo.
3. Recupera la **lista de MovimientoCaja** asociados a ese turno, ordenados cronológicamente (por fechaHoraMovimiento). Incluye datos como tipo (ingreso/egreso), monto, descripción, referencia de comprobante si la hay, etc.
4. *Output:* Devuelve esa lista de movimientos, posiblemente formateada como una tabla o un JSON con un array de movimientos. Cada elemento incluirá sus detalles (ejemplo de salida simplificada: `[{ tipo: "INGRESO", monto: 100.0, descripcion: "Venta FAC-123" }, { tipo: "EGRESO", monto: 10.0, descripcion: "Compra insumos" }, ... ]`). En UI se mostraría un listado con la suma de ingresos, suma de egresos y saldo neto.

#### Flujos Alternativos:

- **E1: Turno no encontrado** – Si el `turnoId` no corresponde a ningún turno (o no pertenece a la empresa/tenant actual), se devuelve error “Turno no encontrado”.
- **E2: Acceso denegado** – (No mencionado en la fuente, pero lógicamente) si un cajero intenta ver el historial de un turno que no es el suyo y no está cerrado, podría negarse por seguridad. Un auditor con permisos vería todo. Este chequeo de permisos ocurriría antes de mostrar datos.
- **E3: Sin movimientos** – Si el turno existe pero no tiene movimientos (posible si fue recién abierto), el sistema puede devolver lista vacía o mensaje “No hay movimientos registrados aún”. Esto no es error, solo un caso borde informativo.

**Eventos Generados:** Ninguno (consulta pasiva).

**Entidades Afectadas:** Ninguna (solo lectura de MovimientoCaja asociados a TurnoCaja).

#### Diagrama de Secuencia (Consulta Historial):



### 2.6 Descarga de Reporte de Cierre

**Actor Principal:** Cajero (que cerró la caja) o Gerente.

**Descripción:** Permite obtener el reporte (en formato PDF u otro) generado al cierre de un turno, el cual contiene el resumen del turno: saldo inicial, todos los movimientos, saldo final, diferencias, etc., generalmente para archivo o impresión.

**Precondiciones:** El TurnoCaja debe estar **CERRADO** para que exista un reporte final de cierre. (No tendría sentido generar el reporte antes de cerrar, aunque podría haber un pre-report). Además, debe haberse generado efectivamente el reporte; asumimos que siempre que se cierra se genera.

#### Flujo Principal:

1. *Input:* El actor selecciona un turno cerrado y solicita descargar su reporte de cierre. Esto puede ser un botón “Descargar PDF” asociado al turno en la interfaz.
2. El sistema recopila nuevamente los datos del TurnoCaja y sus movimientos, o más comúnmente, busca si ya existe un archivo de reporte generado. En muchos casos, en el momento de cierre el PDF ya fue creado y almacenado en alguna ubicación (por ejemplo, en base de datos o sistema de archivos).
3. Si el reporte no existía (por alguna razón), el sistema compila los datos: lee TurnoCaja (saldo inicial, final, diferencia), sus movimientos, y formatea un **documento PDF** con esa información. Esto implica usar quizás una plantilla con encabezado (empresa, usuario, fecha) y cuerpo con una tabla de movimientos y sumas. (Este paso probablemente invoca un componente de infraestructura, no detallado en dominio).
4. *Output:* El sistema proporciona un **enlace de descarga** o inicia la descarga del archivo PDF resultante. Para el actor, el resultado es que obtiene el archivo del reporte en su dispositivo.

#### Flujos Alternativos:

- **F1: Turno no cerrado** – Si se intenta solicitar un reporte de un turno aún abierto, el sistema puede rechazar con error “El turno aún no se ha cerrado, no hay reporte disponible”. (Precond. ya lo evitaba, pero podría darse por error de usuario).
- **F2: Error en generación** – Si ocurre un fallo al generar el PDF (por ejemplo, problema de sistema, o plantilla no encontrada), se notifica al usuario “Error al generar reporte, intente nuevamente más tarde”. El flujo termina con error pero el turno sigue cerrado (no afecta datos).
- **F3: Reporte ya descargado** – No es propiamente un alterno error, pero el sistema podría simplemente volver a dar el enlace sin regenerar nada si ya estaba generado.

**Eventos Generados:** Ninguno (la generación del reporte en sí no emite eventos de dominio, es un artefacto).

**Entidades Afectadas:** Ninguna del modelo de dominio (el reporte es un producto de aplicación; no obstante, podría registrarse un Domain Event o log de “ReporteCierreGenerado” si se quisiera llevar control de descargas, pero no es usual).

#### Diagrama de Secuencia (Descarga de Reporte):



## 2.7 Anulación de Movimiento de Caja

**Actor Principal:** Cajero (de ser un movimiento manual propio) o Supervisor (en casos que se requiera autorización).

**Descripción:** Revierte o elimina un movimiento de caja registrado por error o por anulación de una transacción. Por ejemplo, si se ingresó dos veces un pago por equivocación, se “anula” uno; o si una venta se anuló manualmente sin evento automático, el cajero podría anular el ingreso correspondiente. **Nota:** Los movimientos automáticos asociados a comprobantes normalmente se anulan vía el evento **ComprobanteAnulado** (flujo externo); este caso de uso cubre principalmente movimientos manuales mal ingresados o situaciones excepcionales.

#### Precondiciones:

- Debe existir el **MovimientoCaja** que se desea anular, y su turno asociado debe estar **ABIERTO** (no se permiten alteraciones después del cierre).
- Si el movimiento a anular fue generado automáticamente (origen **EVENTO\_COMPROBANTE**), probablemente solo un Supervisor pueda anularlo manualmente, o preferiblemente, se espera el evento del sistema de facturación. Si un cajero intenta anular un movimiento automático, podría negarse.

- El usuario debe tener permiso para anular (posiblemente *AnularMovimientoCaja* para Supervisor).

#### **Flujo Principal:**

1. *Input:* El actor selecciona un movimiento existente (por su `movimientoId`) que desea anular.
2. El sistema verifica: que dicho movimiento pertenezca a un turno abierto y que no haya sido ya anulado antes. Si el movimiento tiene origen automático y la política no permite anularlo manualmente, se podría frenar aquí (o requerir credencial de Supervisor).
3. Para anular, el sistema crea un **MovimientoCaja “inverso”**:
  - Si el movimiento original era un INGRESO de X monto, se crea un nuevo movimiento EGRESO de igual monto y con descripción indicando que es anulación del movimiento tal.
  - Si era un EGRESO, se crea un INGRESO de igual valor (aunque egresos manuales raramente se “anulan”, podría darse si se reversionó un gasto).

El nuevo movimiento inverso se asigna al mismo turno. Básicamente, en vez de borrar, se contrarresta su efecto.
4. El *TurnoCaja* recalculará su *saldoActual* sumando este nuevo movimiento (que netea la suma del original). Así, el saldo vuelve al que tenía antes del movimiento anulado. El sistema podría marcar el movimiento original como “Anulado” (un flag o estado), para no mostrarlo como activo en reportes (o al menos indicarlo).
5. Se **publica un evento** `MovimientoCajaRegistrado` para el nuevo movimiento (el de anulación). No necesariamente hay un evento distinto de “MovimientoAnulado” a menos que se quiera; en muchos diseños, basta con registrar un nuevo movimiento de signo opuesto.
6. *Output:* Se notifica al usuario que el movimiento fue anulado exitosamente. En la interfaz, el movimiento original podría aparecer tachado o marcado como anulado, y uno nuevo “anulación de X” añadido a la lista.

#### **Flujos Alternativos:**

- **G1: Movimiento irreversible** – Si el movimiento no puede anularse (por ejemplo, ya fue anulado antes, o pertenece a un turno cerrado, o es automático no permitido anular manual), el sistema lanza un error de validación “Este movimiento no puede anularse”. *Ejemplo:* intentar anular un ingreso generado por una factura sin ser Supervisor resultaría en error.
- **G2: Permiso insuficiente** – Si un cajero normal intenta anular un movimiento que por políticas requiere Supervisor (p. ej. un movimiento de salida de efectivo grande), se rechaza con mensaje “Requiere autorización de Supervisor”.
- **G3: Anulación de comprobante ya manejada** – Si el movimiento a anular proviene de una venta que ya se anuló vía evento, podría no haber nada que hacer (el sistema podría decir “la anulación ya fue procesada automáticamente”).

**Eventos Generados:** `MovimientoCajaRegistrado` (para el movimiento inverso). (Eventualmente podría existir un evento específico `MovimientoAnulado` pero no es estrictamente necesario si se comunica con el registro del nuevo movimiento).

**Entidades Afectadas:** Crea un nuevo **MovimientoCaja** (inverso). Marca/actualiza el **MovimientoCaja original** como anulado (esto puede ser simplemente un campo `anulado=true` o asociarlo con el movimiento inverso). Actualiza el **TurnoCaja.saldoActual**.

#### **Diagrama de Secuencia (Anulación de Movimiento):**



## 2.8 Ajuste de Saldo por Discrepancia

**Actor Principal:** Supervisor (o Gerente).

**Descripción:** Este caso de uso ocurre cuando, durante el cierre de turno, se detectó una discrepancia mayor a la tolerada (ver flujo alterno C1 del cierre). El Supervisor revisa la situación y realiza un ajuste en el registro para resolver la diferencia: puede ser recontar el efectivo o autorizar registrar la pérdida/ganancia como un movimiento extra. El resultado es que la diferencia quede dentro de límites para poder cerrar.

**Precondiciones:**

- Debe haber un TurnoCaja **ABIERTO** que intentó cerrarse y quedó pendiente por discrepancia, es decir, con **diferenciaCaja** calculada y marcada como fuera de tolerancia (una especie de “estado no cerrado, en ajuste”). Alternativamente, el sistema no cambia de estado pero simplemente no cerró; sabemos que hay diferencia > tolerancia.
- Solo un usuario con rol Supervisor (o similar) puede ejecutar este ajuste (permiso *AjustarSaldo*).
- El monto de efectivo original contado está registrado; se necesitará un nuevo valor contado tras corrección.

**Flujo Principal:**

1. **Input:** El Supervisor ingresa un **nuevo valor de efectivo contado** tras realizar un re-conteo o corrección.

Básicamente dice: “En realidad en caja hay X dinero” donde X difiere del valor inicial que dio el cajero. Esto podría ser porque el cajero contó mal y ahora con el supervisor recontaron.

2. El sistema recalcula la **diferenciaCaja** con este nuevo **efectivoContado**. Ahora idealmente la diferencia queda <= tolerancia. (Si aún excede, se podría repetir conteo o ya entrar a registrar la pérdida, pero supongamos que con el nuevo valor ya entra en rango).

3. El TurnoCaja se actualiza con el nuevo **efectivoContado** y la nueva **diferenciaCaja**. **Opcional:** el supervisor podría en este paso decidir inmediatamente **cerrar** el turno si ya está conforme, desencadenando el flujo de cierre (caso 3) a continuación. O bien, si la política exige, primero registra la discrepancia oficialmente y luego cierra.

4. (Si la herramienta prevé un registro explícito:) El sistema puede ahora permitir completar el cierre. Podríamos integrar este caso 8 con el 3: tras ajuste exitoso, se llama internamente al flujo de cierre (paso 3 en adelante del caso 3.3). De hecho, en el diagrama del caso 3.3 se ve que tras el ajuste se recalcula diferencia y se prosigue con cierre.

5. **Output:** Si el ajuste fue solo un re-conteo, no genera salida visible aparte de permitir el cierre. Quizás se muestra un mensaje “Diferencia ajustada, proceda a cerrar”. En la práctica, tras ajuste, el Supervisor probablemente ejecuta la acción de cierre (Caso 3) y ese tendrá su confirmación. Si modelamos el ajuste como separado, podríamos decir que la salida es una confirmación tipo “Saldo ajustado. Diferencia actual: S/0.50 (dentro de tolerancia). Puede cerrar la caja.”.

**Flujos Alternativos:**

- **H1: Monto inválido** – Si el Supervisor ingresa un valor ilógico (negativo, o menor al total de ingresos lo cual no tiene sentido físico a menos que haya dinero faltante extremo – aunque puede haberlo; -10 significaría pérdida de 10?), el sistema valida y rechaza con “Monto de efectivo inválido”.
- **H2: Diferencia aún excesiva** – Si incluso tras un ajuste la diferencia sigue > tolerancia (quizá el supervisor no quiso ajustar del todo), se mantiene el bloqueo. Se podría iterar: el sistema informa “Aún existe diferencia de X, ajuste adicional requerido”. Quizá en ese caso, el supervisor decide registrar la diferencia entera como pérdida: en lugar de cambiar efectivo, podría optar por registrar un movimiento de **Ajuste** por la diferencia. Ese sería otro camino: en lugar de recontar, se ingresa un movimiento de ajuste (lo que conceptualmente es distinto: añades un movimiento para cuadrar). El caso 8 podría abarcar también:

- *Camino alterno*: Supervisor crea un movimiento extraordinario (por ejemplo, un egreso por el faltante, con motivo “Pérdida al cierre”). Eso se registraría similar al caso 7 pero específico al cierre. Luego la diferencia quedaría en 0, permitiendo cerrar.
- **H3: Permiso insuficiente** – Si alguien sin rol intenta este flujo (no debería, pero por completitud), se niega con “Operación solo para Supervisor”.

**Eventos Generados:** Si el ajuste implica registrar un movimiento, entonces se generaría un `MovimientoCajaRegistrado` por ese movimiento de ajuste. Si solo fue re-calcular y cerrar, no hay evento propio del ajuste, solo el eventual `TurnoCajaCerrado`. Se podría definir un evento interno `DiferenciaCajaAjustada` si se quisiera auditar la intervención, pero no es estrictamente necesario.

**Entidades Afectadas:** Actualiza el `TurnoCaja` (cambia efectivoContado y diferencia antes del cierre). Posiblemente crea un `MovimientoCaja` de ajuste (si esa vía se toma).

#### Diagrama de Secuencia (Ajuste de Saldo):



#### 2.9 Delegar Cierre de Turno

**Actor Principal:** Supervisor.

**Descripción:** Permite que un Supervisor cierre un turno que **fue abierto por otro cajero** que, por alguna razón, no puede cerrarlo (ausencia, cambio de turno, emergencia, etc.). Esto asegura que no queden turnos abiertos indefinidamente y que un encargado pueda conciliarlos. Es básicamente el mismo proceso de cierre pero iniciado por un usuario distinto.

#### Precondiciones:

- Debe existir un `TurnoCaja ABIERTO` perteneciente a otro usuario/cajero. (Si el turno es del mismo usuario, no es delegación, sería caso normal).
- El Supervisor debe tener permiso `CerrarCajaDelegada` (o ser de rol que lo incluya).
- Idealmente, el cajero original no debería estar activo en ese momento (por política interna, no técnica).

#### Flujo Principal:

1. *Input:* El Supervisor elige el turno abierto de un cajero (identificado por `turnoId` o por caja y usuario) que desea cerrar. Indica la acción de cierre delegada.
2. El sistema verifica el permiso del Supervisor para esta acción. Si no lo tiene, rechazo (flujo alterno).
3. A continuación, el sistema ejecuta esencialmente el **mismo flujo de cierre** descrito en 3.3 (pasos 1 a 5) para ese turno: solicita el efectivo contado (posiblemente el Supervisor debe recabar cuánto dinero dejó el cajero en caja; puede contarlo él mismo), calcula diferencia, etc., y cierra el turno. La única diferencia es que se marca que el cierre fue realizado por otro usuario. En la implementación, esto puede reflejarse solo en auditoría; el `responsableId` del turno normalmente queda siendo el cajero original y así se emitirá en el evento `TurnoCajaCerrado`, pero se podría añadir un campo “cerradoPor” en el evento para indicar que lo cerró un supervisor (en este caso, la instrucción en [21] L2046-L2054 sugiere que el evento incluye el `responsableId` original, manteniendo la trazabilidad de quién era el dueño del turno).
4. Se publica el evento `TurnoCajaCerrado` con el `responsableId` original (no el del supervisor), de modo que otros contextos sepan que cerró el turno del cajero X. (Si se quisiera, se podría extender el evento o lanzar otro evento “`TurnoCajaCerradoDelegado`” para notificar al cajero original o algún log, pero no es estrictamente necesario).
5. *Output:* El Supervisor recibe confirmación de que el turno de fulano ha sido cerrado con éxito. Posiblemente se genera/imprime el reporte de cierre igual que siempre, que el supervisor puede firmar o entregar al área

correspondiente. El cajero original podría recibir una notificación de que su caja fue cerrada por Supervisor (pero eso es fuera de este contexto, más en notificaciones generales).

#### Flujos Alternativos:

- **I1: Permiso insuficiente** – Si quien intenta la delegación no tiene privilegios, el sistema rechaza con “No tiene permiso para cerrar turnos de otros usuarios”. Solo usuarios con rol adecuado pueden.
- **I2: Turno ya cerrado** – Si por error se intenta delegar cierre de un turno que ya fue cerrado (quizá por el cajero mismo), se notifica que no es posible (turno no está abierto).
- **I3: Diferencia excesiva** – Si al proceder con el cierre delegado se encuentra diferencia fuera de tolerancia, se aplica el caso 8 (ajuste) de igual manera. El supervisor en este caso ya está presente, así que él mismo ejecutaría el ajuste y concluiría.
- En resumen, fuera del iniciador, todas las validaciones de cierre aplican igual.

**Eventos Generados:** TurnoCajaCerrado (idéntico al normal, indicando el cierre del turno del cajero original).

**Entidades Afectadas:** TurnoCaja del cajero original es actualizado a cerrado; Movimientos se finalizan; reporte generado. El hecho de que fue delegación puede o no reflejarse en los datos (depende si se quiere guardar quién cerró; podría guardarse en un campo “cerradoPor = supervisorId” en TurnoCaja para registro).

#### Diagrama de Secuencia (Cierre Delegado):



Estos nueve casos de uso cubren las operaciones principales dentro del Bounded Context **ControlCajaBC**, demostrando cómo se orquesta la apertura, operación diaria y cierre de la caja, incluyendo escenarios excepcionales y roles de usuarios diferenciados.

## Eventos de Dominio\_cc

En ControlCajaBC se generan y consumen diversos eventos de dominio tanto internos (propios del contexto) como externos (provenientes o dirigidos a otros contextos). A continuación se listan y documentan cada evento relevante, indicando su payload, quién lo produce, quién lo consume y un ejemplo de contrato de mensaje en formato JSON:

### 4.1 Lista de Eventos Internos y Externos ↗

#### Eventos Internos (emitidos por ControlCajaBC):

- **TurnoCajaAbierto** – *Productor:* ControlCajaBC (emitido al abrir un turno). *Consumidores:* IndicadoresNegocioBC (registra inicio de turno en métricas), Módulo de UI/notificaciones (habilita interfaz de caja abierta para el usuario). *Payload:* `turnoId`, `responsableId` (cajero que abrió), `codigoCaja`, `fechaHoraApertura`. *Contrato JSON:* ya mostrado en sección 2.1.5, e.j.: `"event": "TurnoCajaAbierto", "turnoId": "...", ...`. Este evento permite a otros contextos saber que existe un nuevo turno de caja operando, lo que inicia conteos en dashboards (p.ej., contar 1 turno activo) y puede activar funcionalidades (por ejemplo, no permitir más de uno abierto por caja en orquestador, aunque eso se previene internamente).
- **MovimientoCajaRegistrado** – *Productor:* ControlCajaBC (cada vez que se registra un movimiento, manual o automático). *Consumidores:* IndicadoresNegocioBC (actualiza estadísticas de ingresos/egresos, montos de venta, etc.); también un sistema de Auditoría (para registro contable detallado). *Payload:* `movimientoId`, `turnoId`, `tipoMovimiento`, `monto`, `fechaHoraMovimiento`. Puede extenderse con detalles como `origenMovimiento` o referencia de comprobante, pero otros contextos posiblemente no necesiten tanto detalle, solo totales. *Contrato JSON:* ejemplo dado en sección 2.1.5. Este evento informa de cada entrada o salida de efectivo, permitiendo por ejemplo a IndicadoresNegocioBC recalcular el **ticket promedio** o total de ventas en tiempo real sumando los ingresos registrados.
- **TurnoCajaCerrado** – *Productor:* ControlCajaBC (al cerrar un turno exitosamente). *Consumidores:* IndicadoresNegocioBC (incluye la información de cierre en reportes de conciliación, posiblemente generando alertas si hubo diferencias); módulo de Reporting PDF central (si existe, aunque normalmente el PDF ya se generó internamente); Orquestador o Servicio de notificación (podría informar a contabilidad que la caja del día X fue cerrada). *Payload:* `turnoId`, `responsableId` (cajero original), `saldoFinal`, `diferenciaCaja`, `fechaHoraCierre`. Ejemplo JSON en sección 2.1.5. Este evento resume el resultado del turno. Otros contextos lo utilizan, por ejemplo IndicadoresNegocioBC puede calcular métricas por turno (ventas por turno, diferencias ocurridas, generar alertas de auditoría si `diferenciaCaja != 0`). También el contexto de notificaciones podría enviar un email al dueño de negocio si hubo descuadre.

#### Eventos Externos (consumidos por ControlCajaBC de otros BCs):

- **ComprobanteEmitido** – *Productor:* ComprobantesElectonicosBC (cuando se genera una factura/boleto electrónica). *Consumidor:* ControlCajaBC (lo consume para registrar un ingreso de caja automático). *Payload:* `comprobanteId`, `montoTotal`, `tipoPago`, `fechaHoraEmision`, etc.. *Contrato JSON:* por ejemplo: `{ "event": "ComprobanteEmitido", "comprobanteId": "FAC-1001", "montoTotal": 350.50, "tipoPago": "EFECTIVO", "fechaHora": "2025-07-04T18:30:00Z", ... }`. Al recibirse, ControlCajaBC verifica el tipo de pago: si es “EFECTIVO”, registra MovimientoCaja de ingreso; si fuera “TARJETA” u otro medio no-caja, quizás no registra nada (o registra un ingreso marcado como no efectivo, dependiendo del

alcance de caja – asumiendo caja = efectivo). Esta integración asegura que *cada venta en efectivo* se refleje en la caja.

- **ComprobanteAnulado** – *Productor:* ComprobantesElectonicosBC (cuando se anula una factura/boleto previamente emitida). *Consumidor:* ControlCajaBC (registra la salida de efectivo correspondiente, revirtiendo el ingreso). *Payload:* comprobanteId , motivoAnulacion , posiblemente monto (aunque no siempre se envía monto, se podría obtener por ID). *Ejemplo JSON:* { "event": "ComprobanteAnulado", "comprobanteId": "FAC-1001", "motivo": "Error en datos", "fechaHora": "2025-07-04T19:00:00Z" } . Al recibirse, ControlCajaBC busca si el comprobante estaba asociado a algún movimiento (p.ej. guarda mapa comprobante→movimiento) y crea el movimiento de egreso reverso. De este modo, si un documento de venta es anulado, la caja queda cuadrada restando ese ingreso.
- **ClienteActualizado** – *Productor:* GestionClientesBC (cuando cambian datos importantes de un cliente, e.g. se modifica su límite de crédito o estado de morosidad). *Consumidor:* ControlCajaBC (lo usa para futuras validaciones al registrar ingresos a cuenta de ese cliente). *Payload:* clienteId , estadoCliente (ACTIVO/INACTIVO), limiteCreditoNuevo , formaPagoPreferida , etc. *Ejemplo JSON:* { "event": "ClienteActualizado", "clienteId": "CL-203", "estado": "INACTIVO", "limiteCredito": 500.00 } . *Uso:* Si un cliente se vuelve INACTIVO o reducen su crédito, ControlCajaBC al intentar registrar un ingreso a cuenta podría rechazar si sobrepasa el nuevo límite o si el cliente está bloqueado (p.ej., no permitir “ventas a crédito” si cliente inactivo). Este evento actúa como *política de actualización* de reglas en ControlCajaBC.
- **PermisosUsuarioActualizado** – *Productor:* ConfiguracionSistemaBC (cuando se cambian los roles/permisos de un usuario del sistema). *Consumidor:* ControlCajaBC (ajusta internamente las autorizaciones para ese usuario en las operaciones de caja). *Payload:* usuarioId , roles : [...], permisos : [...] (lista actualizada). *Ejemplo JSON:* { "event": "PermisosUsuarioActualizado", "usuarioId": "user-12345", "roles": ["Cajero"], "permisos": ["AbrirCaja", "RegistrarMovimiento"] } . Al recibirla, ControlCajaBC podría actualizar la entidad ResponsableCaja del turno abierto si corresponde, o simplemente tener en cuenta en futuras validaciones. Por ejemplo, si a mitad de un turno a un usuario se le quita permiso de cerrar caja (quizá por algún motivo disciplinario), este contexto debería impedir en adelante que ese usuario cierre (forzando delegación). Esto se logra consultando la información actualizada de permisos al momento de la acción, o almacenando el cambio en alguna estructura en ControlCajaBC.

(Nota: Además de estos, existe un evento **PagoExternoRecibido** mencionado en la documentación original, producido por ControlCajaBC hacia GestionClientesBC para registrar en el historial del cliente un pago realizado 【3+L139-147】 【3+L143-151】 . Ese evento ocurre posiblemente cuando se registra un movimiento en caja asociado a un cliente (por ejemplo, un ingreso por pago de cuota). Sería *Outbound* de ControlCajaBC: *Productor:* ControlCajaBC, *Consumidor:* GestionClientesBC (evento PagoExternoRecibido con campos clienteId , monto , fechaPago 【3+L139-147】 . No fue solicitado en el enunciado enumerarlo, pero es parte de la relación con Clientes).\*)

A continuación, se presenta una tabla resumen de los **contratos de los eventos** más importantes, con su origen y uso:

Evento	Origen (Productor)	Consumidor(es)	Descripción (Payload)
TurnoCajaAbierto	ControlCajaBC (al abrir)	IndicadoresNegocio BC, UI	turnoId , responsableId , codigoCaja , fechaHoraApert

			ura . Informa inicio de turno.
<b>MovimientoCajaRegistrado</b>	ControlCajaBC (al mov.)	IndicadoresNegocio BC, Auditoría	movimientoId , turnoId , tipoMovimiento , monto , fechaHoraMovimiento . Notifica registro de ingreso/egreso.
<b>TurnoCajaCerrado</b>	ControlCajaBC (al cierre)	IndicadoresNegocio BC, Reportes	turnoId , responsableId , saldoFinal , diferenciaCaja , fechaHoraCierra . Indica cierre de turno y resultados.
<b>ComprobanteEmitido</b>	ComprobantesElectronicosBC	ControlCajaBC	comprobanteId , montoTotal , tipoPago , fechaHora . Genera ingreso en caja (según tipoPago).
<b>ComprobanteAnulado</b>	ComprobantesElectronicosBC	ControlCajaBC	comprobanteId , motivoAnulacion , fechaHora . Revertir movimiento en caja correspondiente.
<b>ClienteActualizado</b>	GestionClientesBC	ControlCajaBC	clienteId , estado , limiteCredito , etc.. Actualiza referencias para validar ventas a crédito.
<b>PermisosUsuarioActualizado</b>	ConfiguracionSistemaBC	ControlCajaBC	usuarioId , nuevos

			roles/permiso s . Habilita/bloquea apertura/cierre de caja para ese usuario.
--	--	--	---

Como ejemplo final, a continuación se muestra un **JSON de contrato completo** de un evento externo y uno interno, para ilustrar formato y contenido:

- *JSON Ejemplo evento inbound ( ComprobanteEmitido ):*

1 json

CopiarEditar

```
{
  "event": "ComprobanteEmitido",    "comprobanteId": "BOL-2025-000123",    "montoTotal": 120.00,    "tipoPago": "EFECTIVO",    "moneda": "PEN",    "fechaHora": "2025-07-04T15:45:00-05:00",    "clienteId": "CL-500",    "detalle": "Venta mostrador" }
```

*Interpretación:* Es una boleta emitida por S/120 en efectivo, que ControlCajaBC consumirá para registrar S/120 ingreso. Incluye clienteId opcional, etc.

- *JSON Ejemplo evento outbound ( TurnoCajaCerrado ):*

1 json

CopiarEditar

```
{
  "event": "TurnoCajaCerrado",    "turnoId": "550e8400-e29b-41d4-a716-446655440000",    "responsableId": "user-12345",    "saldoFinal": 1570.00,    "diferenciaCaja": 0.00,    "fechaHoraCierre": "2025-07-04T20:00:00-05:00" }
```

*Interpretación:* Turno cerrado sin diferencias, saldo final 1570. Otros sistemas al recibirla podrían apuntar este valor en reportes diarios y ver que no hubo faltantes/sobrantes (diferencia 0).

En conclusión, la coordinación de eventos en ControlCajaBC sigue un **estilo reactivo (event-driven)** donde:

- Se **consumen eventos** de ventas, clientes y configuración para adaptar el comportamiento interno (registrar ingresos automáticos, adecuar validaciones), usando un contrato **conformista** en el caso de los comprobantes (se adoptan los datos tal cual vienen de ComprobantesElectonicosBC).
- Se **publican eventos** de caja que notificarán a contextos interesados, aislando así a ControlCajaBC (no necesita llamar directamente a servicios de Indicadores u otros, simplemente lanza eventos que ellos entienden).

## Restricciones y Reglas de Negocio\_cc

ControlCajaBC implementa varias **restricciones** (condiciones que impiden realizar una acción si no se cumplen) y **reglas de negocio** (lineamientos lógicos que deben mantenerse durante la operación). A continuación se detallan las más relevantes, combinando las invariantes mencionadas y algunas adicionales, con su justificación:

### Restricciones Operativas (Invariantes duras):

- **RES-01: Unicidad de turno abierto:** Solo **un TurnoCaja** puede estar abierto por cada estación de caja a la vez.  
*Justificación:* Evita confusión y sobreposición de registros. Garantiza que cada punto de venta tenga a lo sumo un cajero operando simultáneamente bajo un mismo código de caja.
- **RES-02: Prohibición de movimientos sin turno:** No se puede registrar ningún **MovimientoCaja** si no hay un turno abierto vigente.  
*Justificación:* Un movimiento de caja siempre debe pertenecer a un turno; esta restricción fuerza la apertura formal antes de cualquier transacción, asegurando trazabilidad (sin turno, no hay contexto para el movimiento).
- **RES-03: Validez de referencia en movimientos automáticos:** Cualquier movimiento generado automáticamente debe referenciar un **comprobante válido** (ej. factura/boleta).  
*Justificación:* Previene entradas fantasma; un ingreso marcado como venta debe tener un ID de venta existente, caso contrario es rechazado. Esto conecta consistentemente ventas con caja.
- **RES-04: Inmutabilidad de movimientos automáticos:** Los **movimientos** originados por eventos de venta (automáticos) **no pueden ser alterados o eliminados manualmente**.  
*Justificación:* Protege la integridad de la información proveniente de sistemas oficiales (facturación). Si hubo error en la venta, debe anularse desde el sistema de comprobantes, lo que generará el evento correspondiente; el cajero no puede simplemente borrar un ingreso de venta sin ese rastro. Esto evita encubrimientos de dinero (ej. no poder borrar una venta para justificar falta de dinero).
- **RES-05: Horario para cierre:** El **cierre de turno** debe realizarse dentro del horario operativo configurado para la empresa.  
*Justificación:* Ayuda a cumplir políticas internas (por ejemplo, todas las cajas deben cerrarse al final del día laboral). Si alguien intenta cerrar muy tarde o fuera de turno, podría ser indicativo de una irregularidad o error. Esta restricción podría implementarse como aviso o impedimento total según la criticidad.

### Reglas de Negocio (validaciones/cálculos):

- **RN-CB-001:** El **saldoInicial** al abrir debe ser **mayor o igual a cero**, y preferiblemente coincidir con el **saldoFinal del turno anterior** en esa caja.  
*Significado:* No se puede iniciar caja con saldo negativo (sería ilógico). Además, si el último cierre dejó X dinero, se espera que el cajero cuente ese mismo X al abrir (en caso de cajas continuas), garantizando continuidad de fondos. Si no coincide, podría requerir registro de un ajuste de apertura.
- **RN-CB-002:** El **saldoFinal** de turno se define como  $\text{saldoInicial} + \sum \text{ingresos} - \sum \text{egresos}$ , y debe igualar el **efectivoContado** al cierre dentro de la tolerancia permitida.  
*Significado:* Es la fórmula fundamental de cuadratura; garantiza conservación de dinero. Cualquier discrepancia es la **diferenciaCaja**. Esta regla es verificada en el momento de cierre. Durante la operación, la fórmula se usa para calcular **saldoActual** en todo momento.
- **RN-CB-003:** Solo el **responsable original del turno** puede cerrarlo. Si otra persona necesita cerrarlo, debe existir una **delegación explícita** (permiso de Supervisor).  
*Significado:* Responsabiliza al cajero de su caja. Un supervisor no puede arbitrariamente cerrar sin ese protocolo. Esto se implementa mediante roles/permisos (como vimos, Rol Supervisor con permiso *CerrarCajaDelegada*).

- **RN-CB-004:** Si la **diferenciaCaja** (descuadre) supera la tolerancia configurada, el sistema debe **bloquear el cierre** y generar una alerta, requiriendo intervención manual (p.ej., re-conteo o autorización). *Significado:* No se permite cerrar “en rojo” o “sobrante” significativo sin que alguien más lo valide. Esto protege contra errores graves o posibles fraudes, forzando que un supervisor tome cartas en el asunto antes de concluir.
- (*Pendiente de implementar*) **RN-CB-005:** Evaluar tolerancias dinámicas o adicionales. Por ejemplo, podríamos decir que si el descuadre persiste repetidamente se notifique a gerencia, o diferentes tolerancias según turno (diurno vs nocturno) – la documentación la menciona como futura. Aunque no detallada, apunta a extender las reglas de diferencias de forma más flexible.
- **RN-CB-006 (implícita):** Un **cliente inactivo o sobrecargado** no debe poder generar ingresos a cuenta. *Significado:* Si se intenta registrar un ingreso en caja marcándolo como “pago a cuenta de cliente X”, se debería comprobar que el cliente esté ACTIVO y no haya excedido su crédito. Esta regla proviene de GestiónClientesBC pero se aplica en caja al registrar movimientos de tipo crédito. Se podría formalizar como RN-CB-006: “No registrar cobro a cliente inactivo o excedido en crédito”.
- **RN-CB-007 (implícita):** No eliminar registros históricos. En caja, esto se traduce en la política de anulación a través de movimientos inversos (no se borra un Turno ni un Movimiento, se anula generando otro). Aunque no estaba enumerada, es una práctica: conservar siempre el historial completo.

**Referencias a Reglas en Documentación:** Las RN-CB-001 a RN-CB-004 están explícitamente en la documentación base. Las restricciones RES-01 a RES-05 también. Estas reglas han sido incorporadas en las secciones anteriores en la lógica de casos de uso y modelo.

Para ilustrar, una **tabla de reglas** consolidada:

Código	Tipo	Descripción
<b>RES-01</b>	Restricción	Un turno abierto por caja como máximo (unicidad de turno activo).
<b>RES-02</b>	Restricción	Prohibido registrar movimientos sin tener un turno abierto.
<b>RES-03</b>	Restricción	Movimientos de ventas requieren referencia a comprobante válido.
<b>RES-04</b>	Restricción	Movimientos automáticos (ventas) no editables ni eliminables manualmente.
<b>RES-05</b>	Restricción	Cierre de turno solo en horario permitido (según configuración).
<b>RN-CB-001</b>	Regla de Negocio	Saldo inicial $\geq 0$ y consistente con el saldo final anterior.
<b>RN-CB-002</b>	Regla de Negocio	$\text{SaldoFinal} = \text{saldoInicial} + \text{ingresos} - \text{egresos}$ ; diferencia dentro de tolerancia.
<b>RN-CB-003</b>	Regla de Negocio	Solo responsable cierra turno, salvo delegación explícita (Supervisor).

<b>RN-CB-004</b>	Regla de Negocio	Diferencias > tolerancia bloquean cierre y generan alerta.
<b>RN-CB-005</b>	Regla de Negocio	(Pendiente) Manejo especial de tolerancias dinámicas y escenarios de ajuste.

Estas reglas y restricciones se implementan en forma de **validaciones** en los flujos (por ejemplo, antes de abrir, durante registro de movimiento, al cerrar) y mediante **specifications/policies** transversales, como se discutió. Cada una busca mantener la **integridad** del negocio: evitar estados inconsistentes (como dos cajas abiertas a la vez o saldos que no cuadran) y reforzar controles internos (roles, horarios, referencias válidas).

Por ejemplo, antes de ejecutar la operación “Abrir Turno”, el sistema efectúa:

- Check RES-01 via specification *NoHayTurnoAbiertoMisimaCaja*,
- Check RN-CB-001 (saldoInicial válido) via *MontoPositivoSpec*,
- Check permiso AbrirCaja (parte de RN-CB-003 para apertura) via *UsuarioAutorizadoSpec*.

Así, las reglas están embebidas en la lógica de la aplicación, garantizando que ControlCajaBC funcione dentro de los parámetros que el negocio considera aceptables, aumentando la confiabilidad y transparencia de la gestión de caja. (Cualquier intento de violar estas reglas derivará en errores manejados que previenen la acción o notifican la situación anómala.)

## Lenguaje Ubicuo\_cc

ControlCajaBC (Control de Caja) maneja el flujo de efectivo diario en *Factura Fácil*, por lo que comparte un lenguaje común con los expertos del dominio (dueños de microempresas, cajeros) y los desarrolladores. Este lenguaje ubicuo define con precisión los términos y acciones clave, evitando ambigüedades y asegurando que todos entiendan lo mismo. A continuación se detalla el vocabulario compartido más relevante:

- **Entidades de dominio:**

- *Caja*: representa la **caja física o sesión de caja** de la empresa, que se abre al inicio del día o turno con un monto inicial y se cierra al final. Contiene el historial de movimientos (ingresos y egresos) y calcula el saldo disponible.
- *MovimientoCaja*: cada **movimiento de caja** (ingreso o egreso de dinero) registrado. Incluye detalles como fecha/hora, monto y concepto (p. ej., *venta*, *gasto*, *retiro*). Puede ser un **Ingreso** (aumenta el saldo) o un **Egreso** (disminuye el saldo).
- *Arqueo*: también llamado **cierre de caja**, es el resumen final al cerrar la caja, que valida el efectivo contado vs. el saldo calculado por el sistema. El arqueo confirma si hay sobrante o faltante de dinero al final del periodo.

- **Objetos de Valor:**

- *Monto*: valor monetario encapsulado, que incluye la cantidad en una determinada moneda. Garantiza inmutabilidad y valida que no existan montos negativos en movimientos.
- *Moneda*: indica la divisa utilizada (por ej., *Soles*, *USD*), definida en Configuración del Sistema. La moneda puede ser un objeto de valor para asegurar consistencia en conversiones y formato.
- *Identificador de Caja*: valor único (por ejemplo, código o UUID) que identifica una sesión de caja. Se usa como VO para evitar confusiones con IDs de otros contextos.

- **Comandos (acciones del usuario):**

- *AperturarCaja*: comando para **abrir la caja** iniciando una nueva sesión, especificando el monto inicial de efectivo. Solo puede ejecutarse si no hay otra caja abierta (invariantes de negocio).
- *RegistrarIngreso*: registra en la caja un **ingreso de efectivo**, por ejemplo, cuando se cobra una venta en efectivo. Requiere detalles como monto e identificación de la venta o motivo.
- *RegistrarEgreso*: registra un **egreso de efectivo**, como pagos menores o retiro de dinero para depósito. Incluye el concepto del egreso para auditoría.
- *CerrarCaja*: comando para **cerrar la caja** al final del día o turno. Calcula el total de ingresos vs egresos y genera el arqueo final; después de este comando la caja queda *cerrada* y no acepta más movimientos.

- **Eventos de Dominio:**

- *CajaAperturada*: evento emitido cuando se abre una caja exitosamente. Incluye el ID de caja y monto inicial. Otros contextos (p. ej. Indicadores de Negocio) podrían suscribirse a este evento para registrar la apertura.
- *IngresoRegistrado*: indica que un ingreso de efectivo fue agregado a la caja, proporcionando detalles del monto y referencia (por ej. a una factura pagada). Puede desencadenar actualizaciones en Indicadores

(como aumento de ventas diarias).

- *EgresoRegistrado*: indica que un egreso de efectivo ocurrió, con detalles para control de gastos.
- *CajaCerrada*: evento al cerrar la caja, contiene el saldo final y resultados del arqueo (ej. sobras o faltantes). Este evento podría ser publicado para que otros BC (como *IndicadoresNegocioBC*) calculen métricas diarias, o para generar un comprobante de cierre si aplica.

## 6.2 Ejemplos de uso en conversaciones

Estos términos del lenguaje ubicuo se emplean cotidianamente en conversaciones entre usuarios del negocio y desarrolladores, por ejemplo:

- **Usuario (Administrador de tienda)**: "Necesito *abrir la caja* cada mañana con el saldo inicial correcto y *cerrarla* al final del día. Ayer tuvimos muchos ingresos por ventas."
- **Desarrollador**: "Perfecto. Al *aperturar la caja* el sistema creará una nueva sesión de **Caja** con el monto inicial, y cada venta en efectivo generará un **IngresoRegistrado**. Al final del día, al *cerrar la caja*, se emitirá un evento **CajaCerrada** con el saldo final para su arqueo."
- **Usuario (Cajero)**: "Acabo de registrar un *egreso* de caja para comprar suministros, ¿quedó reflejado en el sistema?"
- **Desarrollador**: "Sí, al *registrar un egreso*, el **MovimientoCaja** queda guardado con su monto y motivo, y el evento **EgresoRegistrado** se genera. Así, el supervisor puede ver inmediatamente la salida de dinero y el nuevo saldo disponible."

En estas conversaciones, tanto el usuario como el desarrollador utilizan términos como *abrir caja*, *IngresoRegistrado* o *saldo* de forma natural. El lenguaje ubicuo actúa como puente, evitando malentendidos y garantizando que todos hablen de lo mismo con las mismas palabras. Esto agiliza las definiciones de requerimientos y asegura que lo implementado en el código refleje exactamente las operaciones del negocio.

## Servicios de Aplicación\_cc

Los **Servicios de Aplicación** de ControlCajaBC son componentes del *capa de aplicación* encargados de orquestar los casos de uso del contexto. En otras palabras, cada vez que el usuario realiza una operación relevante (abrir caja, registrar un movimiento, cerrar caja), un servicio de aplicación coordina el flujo de esa operación de principio a fin. Estos servicios **no contienen lógica de negocio compleja** en sí mismos (que se delega al dominio), sino que manejan la **orquestación de pasos, transacciones** y la interacción con otras capas (repositorios, servicios de dominio, publicación de eventos). Por ejemplo, un servicio de aplicación iniciará una transacción atómica al abrir o cerrar caja, asegurando que tanto la actualización del estado de la *Caja* como la persistencia de los *Movimientos* ocurran de forma consistente. Si alguna operación falla, el servicio de aplicación se encarga de revertir la transacción o lanzar excepciones antes de propagar eventos de dominio.

### 7.1 Descripción de los Application Services ☰

Dentro de ControlCajaBC típicamente se definen uno o varios **Application Services** alineados con los principales *casos de uso*. Por ejemplo, podríamos tener un servicio de aplicación **ControlCajaService** que expone métodos para las operaciones fundamentales: `abrirCaja()`, `registrarIngreso()`, `registrarEgreso()` y `cerrarCaja()`. Cada método del servicio de aplicación cumple las siguientes responsabilidades:

- **Validación y autorizaciones:** verifica que la precondiciones del caso de uso se cumplan (p. ej., no se puede abrir una caja si ya hay una abierta, o solo un usuario autorizado puede cerrar la caja).
- **Coordinación de la lógica de negocio:** invoca a las entidades o *Domain Services* necesarios. Por ejemplo, podría llamar a un método del agregado *Caja* (`caja.agregarIngreso(...)`) para aplicar la lógica de agregar un movimiento de entrada.
- **Manejo de transacciones:** inicia una *transacción* de base de datos (si la persistencia es relacional) o asegura atomicidad en la operación. Todas las operaciones de persistencia de entidades se coordinan dentro de este contexto, para evitar incoherencias.
- **Coordinación con otros contextos:** si la operación requiere interactuar con otros *Bounded Contexts*, el servicio de aplicación orquesta esas interacciones. Por ejemplo, tras *cerrar caja*, podría llamar a un *servicio de dominio* de *IndicadoresNegocioBC* o publicar un evento para que dicho contexto genere un indicador diario.
- **Publicación de eventos de dominio:** una vez completada la operación con éxito, el servicio publica los eventos correspondientes (*CajaAperturada*, *IngresoRegistrado*, etc.) a través del *Event Bus*, para notificar al resto del sistema. En caso de fallo, podría enviar eventos de compensación o simplemente no emitir eventos hasta asegurar la consistencia.

En resumen, los servicios de aplicación **actúan como fachadas** de casos de uso: reciben las solicitudes de la capa de interfaz (por ejemplo, de un controlador REST), invocan al dominio aplicando las reglas de negocio y coordinan la persistencia y mensajería, todo **manteniendo la integridad de la operación**.

### 7.2 Interfaces y contratos sugeridos ☰

Se recomienda que estos servicios de aplicación expongan **interfaces claras** que puedan ser usadas por los adaptadores (por ejemplo, la API REST o un interfaz de usuario). Un posible diseño es definir una interfaz `IControlCajaService` con métodos que correspondan a las acciones del contexto. Por ejemplo, en pseudocódigo Java:

```

2 java
3
4
5 public interface IControlCajaService {
6     ResultadoAperturaCaja abrirCaja(AperturaCajaCommand cmd);
7     ResultadoMovimiento registrarIngreso(RegistroIngresoCommand cmd);
8     ResultadoMovimiento registrarEgreso(RegistroEgresoCommand cmd);
9     ResultadoCierreCaja cerrarCaja(CierreCajaCommand cmd);
10 }

```

En este esquema, cada método recibe un objeto comando con los datos necesarios (siguiendo el **Command Pattern** dentro de la aplicación) y retorna un resultado con la información relevante (podría ser el estado actualizado de la Caja, o un objeto confirmación). Las **interfaces** permiten desacoplar la implementación concreta (por ejemplo, `ControlCajaServiceImpl`) de su definición, facilitando la prueba unitaria (se pueden mockear estos servicios) y la sustitución o extensión futura sin impactar a quienes lo usan.

En términos de **contratos**, cada método debería especificar claramente:

- **Parámetros de entrada** requeridos (por ejemplo, `monto` y `moneda` para un ingreso, o detalles del usuario/turno para apertura/cierre).
- **Efectos de negocio**: qué sucede al invocarlo (p. ej., `abrirCaja` crea una nueva entidad Caja abierta y lanza evento `CajaAperturada`).
- **Resultado devuelto**: confirma la operación (p. ej., objeto `ResultadoAperturaCaja` con el ID de la nueva caja y su estado).
- **Excepciones o errores**: casos en que no se pueda completar (p. ej., error `CajaYaAbiertaException` si se intenta abrir una caja cuando ya hay una abierta).

Esta especificación clara forma un **contrato** entre la capa de aplicación y los clientes de la misma (p. ej., la capa de interfaz), garantizando qué se puede esperar de cada operación.

### 7.3 Ejemplos de métodos, parámetros y respuestas

A continuación, se ilustran ejemplos de métodos del servicio de aplicación, con sus parámetros y posibles respuestas, usando un estilo cercano al código o JSON:

- **Aperturar Caja**: al invocar `abrirCaja(AperturaCajaCommand)`, el comando contendría datos como `{ cajaId, usuarioId, montoInicial }`. Por ejemplo:

```

1 // Petición de apertura de caja (comando)
2 {
3     "cajaId": "CX-20230701-1",
4     "usuarioId": "U-12345",
5     "montoInicial": 500.00,
6     "moneda": "PEN"
7 }

```

La respuesta podría ser un objeto JSON confirmando la apertura:

```

1 json// Respuesta exitosa de abrirCaja
2 {
3     "cajaId": "CX-20230701-1",
4     "estado": "ABIERTA",
5     "fechaApertura": "2025-07-04T09:00:00Z",
6     "saldoInicial": 500.00

```

```
7 }
8
```

Este resultado corresponde al objeto `ResultadoAperturaCaja` mencionado en la interfaz, incluyendo el estado de la caja y datos relevantes.

- **Registrar Ingreso:** mediante `registrarIngreso(RegistroIngresoCommand)`, donde el comando proporciona `{ cajaId, monto, concepto, referenciaFacturaId }`. Ejemplo de entrada:

```
1 {
2   "cajaId": "CX-20230701-1",
3   "monto": 150.00,
4   "concepto": "Cobro de Venta",
5   "referenciaFacturaId": "F-981"
6 }
```

La respuesta (`ResultadoMovimiento`) confirmaría el nuevo movimiento registrado:

```
1 {
2   "cajaId": "CX-20230701-1",
3   "tipoMovimiento": "INGRESO",
4   "monto": 150.00,
5   "concepto": "Cobro de Venta",
6   "saldoActual": 650.00,
7   "fechaHora": "2025-07-04T10:15:30Z"
8 }
```

Aquí se aprecia el **saldoActual** actualizado después del ingreso, indicando que el servicio recalcularó el saldo de la caja correctamente.

- **Cerrar Caja:** al ejecutar `cerrarCaja(CierreCajaCommand)` con datos como `{ cajaId, usuarioId }`, la operación produce un resultado `ResultadoCierreCaja`. Una respuesta posible sería:

```
1 {
2   "cajaId": "CX-20230701-1",
3   "estado": "CERRADA",
4   "fechaCierre": "2025-07-04T18:00:00Z",
5   "totalIngresos": 1200.00,
6   "totalEgresos": 300.00,
7   "saldoFinalCalculado": 900.00,
8   "diferenciaArqueo": 0.00
9 }
```

Este JSON indica que la caja se cerró con éxito, mostrando los totales del día y que no hubo diferencia en el arqueo (saldoFinal vs efectivo contado). Además, tras este resultado, el sistema publicaría el evento *CajaCerrada* para notificar a otros contextos que la caja del día ha concluido.

Estos ejemplos demuestran cómo los servicios de aplicación manejan entradas y salidas de forma estructurada, sirviendo de **puente entre la capa de dominio y las interfaces externas** (p. ej. APIs REST), manteniendo la consistencia de la información que viaja hacia y desde el dominio.

## Infraestructura y Adaptadores\_cc

La implementación de ControlCajaBC sigue una arquitectura hexagonal (puertos y adaptadores), donde los detalles de infraestructura se aíslan de la lógica de dominio. A continuación se describen los principales componentes de infraestructura y adaptadores utilizados en este contexto:

### 8.1 Implementación de componentes de infraestructura

- **Repositorios:** Se aplica el patrón *Repository* para el acceso a los datos de la caja y sus movimientos. La interfaz de *CajaRepository* (por ejemplo) proporciona métodos como `guardar(Caja)`, `obtenerPorId(cajaId)` o `obtenerCajaAbierta()`. En la infraestructura, esta interfaz tiene implementaciones concretas usando tecnologías de persistencia. Por ejemplo, en un entorno relacional se usa JPA/Hibernate con anotaciones *Entity* para mapear la entidad Caja a una tabla en la base de datos. Alternativamente, si los requerimientos lo justifican (por simplicidad o escalabilidad de lecturas), podría usarse una base documental (MongoDB) con un ODM. En cualquier caso, los repositorios encapsulan los detalles de consulta/almacenamiento, devolviendo entidades del dominio a la capa de negocio. La consistencia transaccional se garantiza mediante transacciones locales en la base de datos subyacente al ejecutar operaciones de estos repositorios.
- **Gateways/Adapters de mensajería:** Para integrar ControlCajaBC con el *bus de eventos* del sistema, se implementan adaptadores de mensajería que actúan como **Gateways**. Por un lado, cuando el dominio genera eventos (*CajaAperturada*, *CajaCerrada*, etc.), un componente *EventPublisher* (adapter de salida) los toma y los publica en la infraestructura de mensajería (por ejemplo, un tópico Kafka o un exchange de RabbitMQ). Por otro lado, para suscribirse a eventos relevantes de otros contextos (p. ej., un evento *FacturaPagada* de ComprobantesElectonicosBC), existe un adapter de entrada (por ejemplo, un listener de Kafka) que traduce el mensaje entrante a una acción en ControlCajaBC (podría invocar un caso de uso de *registrarIngreso* automáticamente). Estos adaptadores desacoplan el código de dominio de los detalles de Kafka/RabbitMQ (como brokers, topics, particiones), manejando la configuración de consumidores y productores.
- **Adapters de API (REST/GraphQL/WebSocket):** La interacción de usuarios y sistemas externos con ControlCajaBC ocurre a través de adaptadores de entrada en forma de API. Lo más común es exponer una **API RESTful** para operaciones de caja. Por ejemplo, una petición HTTP POST a `/caja/abrir` invocaría internamente al servicio de aplicación `abrirCaja`. Este adaptador REST transforma JSON HTTP en comandos/DTO entendibles por la aplicación y viceversa, transformando resultados o excepciones a respuestas HTTP (códigos 200, 400, etc.). Adicionalmente, podrían existir otros adaptadores: una API **GraphQL** para consultas flexibles de históricos de caja, o incluso un canal **WebSocket** para notificar en tiempo real la llegada de nuevos movimientos de caja a un dashboard. Cabe destacar el posible uso de un *ChatAdapter*: por ejemplo, integrar el sistema con una interfaz de chat (como un bot de WhatsApp o Telegram) donde comandos de texto como "abrir caja con S/500" sean interpretados y enviados al servicio de aplicación. Esto se logra mediante un adaptador especializado que traduce mensajes de chat en comandos de la aplicación, ampliando las vías de interacción sin modificar el dominio.
- **EventBusAdapter (eventos públicos y suscripciones):** Si bien ya se mencionó el bus de eventos en los gateways de mensajería, a nivel de infraestructura de ControlCajaBC se suele implementar un componente central para manejar el *Event Bus*. Este **EventBusAdapter** configura las suscripciones a eventos públicos de otros contextos y publica los eventos del propio contexto. Por ejemplo, al iniciar, ControlCajaBC se suscribe (vía este adapter) a

eventos de *ComprobantesElectonicosBC* relevantes (como pagos) y los encamina a un *EventHandler* local. Igualmente, cuando se cierra la caja, el adapter publica el evento *CajaCerrada* en el bus para que cualquier servicio suscrito (p. ej. *IndicadoresNegocioBC*) lo reciba. Este adapter puede aprovechar librerías de integración (p.ej., Spring Cloud Streams, MassTransit, etc.) para simplificar el manejo de eventos. El resultado es una **arquitectura orientada a eventos**, donde *ControlCajaBC* emite y responde a eventos distribuidos sin acoplarse directamente a otros contextos.

## 8.2 Patrones aplicados (Factory, Handler, Mapper)

En la construcción de la solución de *ControlCajaBC* se han aplicado varios **patrones de diseño** para mantener un código limpio y flexible:

- **Factory:** Se utiliza el patrón *Factory* para la creación de agregados y entidades complejas. Por ejemplo, una *CajaFactory* podría encargarse de instanciar una nueva entidad *Caja* al abrirse, asegurando que se provean todos los valores requeridos (monto inicial, fecha, responsable, etc.) y que se apliquen reglas iniciales (como saldo = monto inicial). Esto evita lógica de creación dispersa en el código y centraliza en un punto la construcción válida de objetos de dominio.
- **Handler:** En este contexto, *Handler* alude típicamente a *Command Handlers* y *Event Handlers*. Los **Command Handler** son clases (o métodos) responsables de manejar la ejecución de un comando de aplicación específico. Por ejemplo, un *AbrirCajaHandler* tomaría el comando *AperturaCajaCommand* y coordinaría la llamada al dominio (posiblemente usando el servicio de aplicación) para realizar la apertura. Del mismo modo, los **Event Handler** se suscriben a eventos de dominio (o integraciones) para reaccionar en consecuencia. En *ControlCajaBC*, podríamos tener un handler para el evento *FacturaPagada* (de *ComprobantesElectonicosBC*) que invoque *registrarIngreso* automáticamente. La implementación de estos handlers sigue el principio de *Single Responsibility*, separando cada caso de uso o reacción a evento en componentes dedicados.
- **Mapper:** Se emplean *mappers* para convertir entre objetos de distintas capas. Un **DTO Mapper** transforma entidades de dominio (*Caja*, *MovimientoCaja*) en objetos de transferencia de datos para enviar por la API (por ejemplo, convertir a JSON las propiedades relevantes). Igualmente, un **ORM Mapper** (ya sea mediante frameworks como JPA o manualmente) se encarga de mapear los objetos de dominio a estructuras de persistencia (tablas SQL o documentos NoSQL). Gracias a estos mapeadores, el dominio permanece aislado de detalles externos: por ejemplo, la entidad *Caja* no necesita conocer detalles de la base de datos ni del formato JSON de la API, ya que los mappers se ocupan de esa traducción.

Estos patrones, combinados, contribuyen a una implementación alineada con DDD: las *Factories* garantizan consistencia en la creación, los *Handlers* organizan la lógica de casos de uso y reacciones a eventos, y los *Mappers* aíslan las conversiones entre la capa de dominio y las demás capas.

# Mapa de Contextos\_cc

## 9.1 Diagrama y descripción del Context Map

El Context Map de *Factura Fácil* sitúa a **ControlCajaBC** en relación con otros *Bounded Contexts* del sistema, mostrando cómo interactúan y qué tipo de vínculos existen entre ellos. A continuación se presenta un diagrama simplificado seguido de la descripción de cada relación:



### Relaciones entre ControlCajaBC y otros BC internos:

- **ComprobantesElectonicosBC (Facturación electrónica)** – Relación *Customer-Supplier*:  
*ComprobantesElectonicosBC* actúa como *supplier* (proveedor) de eventos de negocio que *ControlCajaBC* consume. Específicamente, cuando en Comprobantes se registra una factura pagada (por ejemplo, una venta cobrada en efectivo), se emite un evento **FacturaPagada** que *ControlCajaBC* suscribe para automáticamente registrar un ingreso de caja. De esta forma, *ControlCaja* no necesita invocar directamente servicios de facturación, sino que reacciona a sus eventos (*integración por coreografía de eventos*). Adicionalmente, existe un contrato implícito: *ControlCajaBC* confía en el formato de evento de Comprobantes (posiblemente adoptando el mismo modelo de datos para el monto pagado, fecha, etc.), actuando aquí como *Conformist* respecto al contexto de facturación. Si hubiese alguna complejidad en la conversión (por ejemplo, distintas monedas o formatos), se podría introducir un **Anti-Corruption Layer (ACL)** entre ambos para traducir los datos de facturas al lenguaje de caja, manteniendo aisladas las lógicas internas. En general, esta integración asegura que cada venta pagada se refleje en caja sin pasos manuales.
- **IndicadoresNegocioBC (Inteligencia de negocio)** – Relación *Publisher-Subscriber* (similar a Customer-Supplier pero inverso): Aquí *ControlCajaBC* es quien **publica eventos** de los cuales *IndicadoresNegocioBC* es consumidor. Cada evento como *CajaCerrada*, *IngresoRegistrado* o *EgresoRegistrado* es enviado al bus de eventos, y *IndicadoresNegocioBC* los suscribe para actualizar KPIs financieros (ventas diarias, gastos, cash flow). Este contexto de indicadores actúa como *downstream* (downstream context) respecto a *ControlCajaBC*: es decir, depende de los datos que caja emite. El vínculo suele ser de tipo **Customer-Supplier**, donde Indicadores es el *customer* que quizás adopta tal cual los datos de caja (posiblemente en un modelo de *Reporting*) sin modificarlos – un caso típico de *Conformist* si no se requiere transformación. No obstante, si Indicadores necesita agregar lógica propia o distintos agregados a partir de estos eventos, podría tratar a *ControlCaja* como Open Host, consumiendo los eventos publicados públicamente.
- **ConfiguracionSistemaBC** – Relación *Conformist/Shared Kernel*: *ControlCajaBC* **consulta configuraciones comunes** del sistema desde este contexto. Por ejemplo, la moneda base, parámetros de redondeo de caja, o el horario de corte diario podrían vivir en *ConfiguracionSistemaBC* y ser leídos por *ControlCajaBC*. La relación aquí es muy acotada: *ControlCaja* es *cliente* que simplemente lee datos (posiblemente vía una API interna o mediante eventos de configuración cargados al iniciar). Dado que se trata de datos compartidos y estáticos, en lugar de implementar un ACL complejo, *ControlCaja* actúa como *Conformist*: adopta los datos tal cual los expone Configuración. Incluso podría considerarse un caso de *Shared Kernel* si ambos contextos comparten ciertos componentes (por ejemplo, una librería común para el manejo de Moneda o Fechas). No hay comportamiento de negocio complejo en esta integración, solo consumo de datos maestros.
- **GestionClientesBC, CatalogoArticulosBC, ListaPreciosBC** – **Sin integración directa**: Estos contextos (gestión de clientes, catálogo de artículos, lista de precios) pertenecen al dominio de ventas/facturación pero *ControlCajaBC*

no interactúa directamente con ellos. ¿Por qué? ControlCaja se enfoca en el dinero en efectivo que ingresa o sale, independientemente de quién es el cliente o qué artículos se vendieron – esa información la maneja principalmente *ComprobantesElectronicosBC* y otros contextos upstream. En todo caso, si ControlCajaBC necesita mostrar algún dato de cliente o producto en un reporte de caja, lo haría a través de *Comprobantes* (que contiene la referencia a cliente y detalle de venta) u otro servicio de consulta transversal. Por lo tanto, en el mapa de contexto señalamos *ninguna relación directa* con estos BC. Cada uno permanece autónomo en su propio subdominio.

#### **Integración con Identidad y Autenticación (Mi Contasis):**

Fuera del subdominio de facturación, *ControlCajaBC* requiere confiar en un mecanismo central de **autenticación y multi-tenant**. *Mi Contasis* es la plataforma de identidad utilizada, y la integración se realiza a través de un **Orquestador de Apps**. En la práctica, esto significa que hay un servicio central (Orquestador) que intercepta las solicitudes de usuarios, verifica credenciales con *Mi Contasis* (Single Sign-On) y luego propaga un **token de autenticación** y el **contexto de tenencia (tenant)** hacia los microservicios de Factura Fácil, incluido *ControlCajaBC*. *ControlCajaBC* no maneja autenticación internamente; simplemente confía en que cada llamado ya viene autenticado y con información del usuario/empresa actual, típicamente en los encabezados o metadatos de la solicitud. Esta separación de preocupaciones asegura que todos los BC (Caja, Clientes, Artículos, etc.) tengan un mecanismo uniforme de seguridad sin duplicar lógica de autenticación. El Orquestador de Apps actúa aquí como un *ACL genérico* frente a *Mi Contasis*, traduciendo las afirmaciones de identidad en algo que los BCs entienden (por ejemplo, un ID de usuario interno, roles, permisos) y asegurando que cada contexto se enfoque en su dominio sabiendo quién es el usuario y a qué tenant (empresa) pertenece.

# Política de Consistencia y Transaccionalidad\_cc

## 10.1 Enfoque de consistencia (eventual vs inmediata) ↗

En el diseño de *Factura Fácil* se adoptó una estrategia de **consistencia eventual**, apoyándose en eventos de dominio y el patrón Saga para manejar operaciones que abarcan varios Bounded Contexts. Esto significa que no se utiliza una transacción distribuida tradicional de dos fases (2PC) entre servicios; en su lugar, cada contexto realiza su parte de la operación de forma independiente y comunica el resultado mediante eventos, asumiendo que eventualmente todas las partes estarán consistentes.

Para lograr esto, cuando ocurre una operación que involucra a ControlCajaBC y a otros contextos (por ejemplo, el registro de un cobro de factura que impacta caja y facturación), **se publica un evento** en lugar de intentar bloquear dos bases de datos en una sola transacción. ControlCajaBC, al ser eventual-consistente, podría momentáneamente no reflejar un cambio hasta recibir el evento correspondiente. Sin embargo, este ligero desfase es aceptable en favor de la **disponibilidad y descentralización** de los servicios.

Hay casos donde se requiere asegurar orquestación: para eso se implementan **Sagas**, que pueden ser *coreografiadas* (basadas en eventos distribuidos) u *orquestadas* (con un servicio coordinador). En general, *Factura Fácil* favorece *coreografía* para flujos simples (los servicios reaccionan a eventos entre sí sin control central) y recurre a *orquestación* cuando el flujo es más complejo o necesita lógica de compensación coordinada. Esto mantiene los sistemas acoplados débilmente, evitando un orquestador gigante innecesario salvo en casos justificados.

## 10.2 Orquestadores, transacciones distribuidas y compensaciones ↗

Para operaciones críticas que involucran múltiples contextos, se han diseñado **sagas orquestadas** específicas. Un ejemplo es la *Saga de Cobro de Venta*: cuando un cliente paga una factura, el Orquestador (podría residir en *ComprobantesElectronicosBC* o ser un servicio aparte) inicia el proceso. Paso a paso:

1. **Registrar factura pagada** (en *ComprobantesElectronicosBC*) – se marca la factura como cobrada y se emite un evento *FacturaPagada*.
2. **Registrar ingreso en caja** (en *ControlCajaBC*) – al recibir el evento, se invoca el caso de uso de registrar ingreso de efectivo. Si este paso tiene éxito, *ControlCajaBC* emitirá *IngresoRegistrado*.
3. **Confirmación** – el orquestador escucha el evento de ingreso registrado; si todo está correcto, puede enviar una confirmación final de cobro aplicado.

Si alguno de los pasos falla, entran en juego los **mecanismos de compensación**. Por ejemplo, si *ControlCajaBC* no puede registrar el ingreso (digamos que la caja estaba cerrada o hubo un error de persistencia), el orquestador detectará la ausencia de confirmación y compensará revertiendo la operación inicial: podría marcar la factura nuevamente como pendiente de cobro o generar un evento de *PagoFallido* para que otros contextos lo manejen. Cada saga define qué compensaciones realizar. En el caso de cierre de caja, si tras emitir *CajaCerrada* se detectara una inconsistencia en *IndicadoresNegocioBC*, no se reabre la caja (eso sería muy complejo), sino que quizás se emite un evento de ajuste o se notifica al admin para corrección manual, dependiendo de la política definida.

Técnicamente, los orquestadores pueden implementarse con herramientas como **State Machines** o motores de workflow integrados, pero en muchos casos un simple servicio escuchando eventos e invocando comandos en orden es suficiente. Lo importante es que *cada paso es una transacción local* en su contexto; al finalizar, publica un evento. El orquestador/saga actúa sobre esos eventos para desencadenar el siguiente paso o la compensación. Esto garantiza

**atomicidad distribuida de manera eventual:** o bien todos los contextos aplican sus cambios, o los que lo hicieron se deshacen mediante compensación, logrando consistencia final sin bloquear sistemas.

En resumen, la política de ControlCajaBC (y en general de Factura Fácil) es **evitar transacciones distribuidas bloqueantes**, optando por una combinación de **eventos, sagas y compensaciones** para mantener la integridad. De este modo se logra resiliencia: cada microservicio puede seguir funcionando independientemente, y ante fallos en la saga, las compensaciones aseguran que no queden inconsistencias permanentes en el negocio.

# Estrategia de Persistencia y Esquema de Datos\_cc

## 11. Estrategia de Persistencia y Esquema de Datos [🔗](#)

### 11.1 Modelo físico de la entidad (tablas o colecciones):

El **Bounded Context ControlCajaBC** persiste su información en un esquema de datos dedicado a las sesiones de caja. La entidad principal es la *Sesión de Caja*, almacenada típicamente en una tabla relacional `CajaSesion`. Cada registro de `CajaSesion` representa una apertura y eventual cierre de caja, incluyendo campos como el identificador único de la sesión, el identificador del tenant (microempresa), el usuario cajero asignado, la fecha/hora de apertura, el monto inicial declarado, la fecha/hora de cierre, el monto final contado y la diferencia calculada entre lo contado y lo esperado. Opcionalmente se almacenan datos calculados como el **total esperado** (suma de ventas en efectivo durante la sesión) para facilitar el cálculo de discrepancias al cierre. A continuación se ilustra un ejemplo simplificado del modelo físico en formato JSON, representando una sesión de caja completa:

```
1 {  
2   "CajaId": "caf9b6b3-8e5a-4f3a-9f19-0011223344aa",  
3   "TenantId": "tenant_123",  
4   "UsuarioId": "user_456",  
5   "SucursalId": "sucursal_001",  
6   "FechaApertura": "2025-07-04T09:00:00Z",  
7   "MontoApertura": 100.00,  
8   "FechaCierre": "2025-07-04T18:00:00Z",  
9   "MontoCierre": 550.00,  
10  "TotalCalculado": 550.00,  
11  "Diferencia": 0.00,  
12  "Estado": "CERRADA"  
13 }
```

### 11.2 Índices recomendados y particionamiento por tenant:

Dado que Factura Fácil es multi-tenant, todas las tablas incluyen un campo `TenantId` para aislar los datos por cliente. Se recomiendan índices compuestos que mejoren el rendimiento de las consultas más frecuentes, filtradas por empresa (tenant) y estado de la caja. Por ejemplo, un índice por (`TenantId`, `Estado`) acelera la obtención de cajas abiertas activas de una empresa, mientras que otro índice por (`TenantId`, `FechaCierre`) optimiza la consulta de historiales de cierre por rango de fechas. En sistemas de base de datos relacional, se puede considerar *particionar horizontalmente* la tabla de sesiones de caja por `TenantId` para distribuir los datos por cliente y mejorar la escalabilidad, especialmente si el número de tenants crece significativamente. Si se utiliza un almacenamiento NoSQL (por ejemplo, colecciones/documentos), se recomienda usar el `TenantId` como parte de la *clave de partición*, garantizando que todos los documentos de una misma empresa residan juntos para consultas eficientes. La siguiente tabla resume algunos índices claves:

Índice	Columnas	Propósito
UX_CajaSesion_Tenant_Abierta	TenantId, UsuarioId <b>(unique)</b>	Garantiza que un cajero tenga solo una sesión abierta a la vez por tenant.

IX_CajaSesion_Tenant_FechaCierre	TenantId, FechaCierre	Acelera la búsqueda de cierres históricos por empresa y fecha.
IX_CajaSesion_Tenant_Estado	TenantId, Estado	Obtención rápida de sesiones abiertas o pendientes por tenant (uso en panel en tiempo real).

### 11.3 Estrategia de versionado y migraciones de esquema:

La evolución del esquema de datos se gestiona mediante migraciones versionadas. Cada cambio en la estructura (nuevas columnas, tablas, etc.) se documenta en scripts de migración incremental numerados según una convención temporal o semántica. Por ejemplo, en entornos .NET se utiliza *Entity Framework Code First Migrations*, manteniendo un historial de migraciones aplicadas en una tabla de control de versiones del esquema. Esto permite que el despliegue de nuevas versiones de ControlCajaBC actualice la base de datos de forma controlada, asegurando compatibilidad hacia atrás cuando sea posible. Además, se sigue una estrategia de *versionado semántico* para los contratos de datos: si se realizan cambios mayores que podrían romper la compatibilidad (por ejemplo, cambiar el tipo de un campo o removerlo), se evalúa introducir una nueva versión de la API o eventos de dominio en lugar de modificar los existentes. De esta forma, los consumidores del contexto (otros bounded contexts o servicios externos) pueden adaptarse gradualmente a los cambios de esquema.

## Seguridad y Autorización\_cc

### 12.1 Autenticación gestionada por Identidad y Autenticación (*Mi Contasis*):

La autenticación de usuarios en Factura Fácil es centralizada. Todas las solicitudes hacia el microservicio ControlCajaBC requieren un **token de seguridad** emitido por la plataforma de identidad *Mi Contasis*, la cual actúa como proveedor de Identidad y Autenticación único para todo el ecosistema. *Mi Contasis* maneja el registro de usuarios, verificación de credenciales y generación de tokens JWT que incluyen los datos del usuario y sus roles autorizados. Cuando un usuario inicia sesión en Factura Fácil (por ejemplo, un cajero o un administrador), obtiene un token firmado por *Mi Contasis*. Dicho token es enviado en cada petición (por ejemplo, en la cabecera HTTP Authorization: Bearer) y validado por el gateway o por el propio microservicio ControlCajaBC, garantizando que solo usuarios autenticados accedan a las operaciones. Este mecanismo delega toda la complejidad de la autenticación en *Mi Contasis*, manteniendo a ControlCajaBC independiente de los detalles de gestión de usuarios y facilitando single sign-on entre los distintos módulos de la plataforma.

### 12.2 Roles y permisos necesarios por operación:

ControlCajaBC define permisos a nivel de **rol de usuario** para proteger cada operación crítica. Los roles principales son: **Rol\_Administrador** (Administrador de la microempresa) y **Rol\_Cajero** (Operador de caja). A continuación, se describen las capacidades según rol:

- **Rol\_Cajero**: puede aperturar una nueva sesión de caja y cerrarla, pero típicamente solo para la caja asignada a sí mismo. También puede registrar movimientos de efectivo durante su turno (p.ej. ingresos por ventas en efectivo recibidas, egresos por cambios o retiros autorizados) si el sistema contempla esta funcionalidad.
- **Rol\_Administrador**: tiene privilegios elevados, incluyendo la capacidad de ver todas las sesiones de caja de la empresa, incluso aquellas abiertas por distintos cajeros. Puede forzar el cierre de una caja en situaciones excepcionales (ej.: cajero olvidó cerrar, discrepancia no resuelta) y revisar los reportes de discrepancias de cierre. Asimismo, el administrador puede configurar usuarios cajeros y sus permisos en *Mi Contasis*, y acceder a cualquier operación de caja para auditoría.

Estas restricciones se implementan tanto en la capa de aplicación como en la de infraestructura. Por ejemplo, al invocar la API de *apertura de caja* se verifica que el usuario posea Rol\_Cajero, y al invocar *consultar cierres históricos* se requiere Rol\_Administrador. La siguiente tabla ejemplifica algunas operaciones clave y los roles requeridos:

Operación	Descripción	Rol requerido
Aperturar caja	Iniciar una nueva sesión de caja en un punto de venta.	Rol_Cajero (usuario asignado) o Rol_Administrador
Cerrar caja	Realizar el cierre, registrando monto final y calcular diferencia.	Rol_Cajero (propietario) o Rol_Administrador (supervisión)
Consultar estado de caja (en curso)	Ver si existe una caja abierta y detalles de la misma.	Rol_Cajero (su propia caja) o Rol_Administrador
Listar/Consultar cierres históricos	Obtener el historial de cierres de caja de la empresa.	Rol_Administrador (todas las cajas)

### 12.3 Contratos de autorización y validaciones en gateways:

La arquitectura implementa un **API Gateway** que orquesta las llamadas a los microservicios, actuando como primer punto de validación de seguridad. Este gateway aplica un contrato de autorización predefinido: antes de enrutar una petición al servicio de ControlCajaBC, verifica que el token JWT provisto por Mi Contasis sea válido, no esté expirado, y contenga los roles adecuados para la operación solicitada. Por ejemplo, si llega una petición para *cerrar caja*, el gateway comprobará que el claim de roles del token incluya Rol\_Cajero o Rol\_Administrador según corresponda; de lo contrario, rechaza la solicitud con error 403 (Forbidden) sin siquiera tocar el microservicio. Adicionalmente, se realizan validaciones de multi-tenant: el gateway puede añadir en cada solicitud el identificador de tenant asociado al usuario (por ejemplo, en un encabezado o claim específico), y el microservicio lo utiliza para filtrar los datos. De esta forma se asegura que un usuario de la Empresa A no pueda acceder accidentalmente a información de la Empresa B, incluso si intentara manipular identificadores en la URL. Estos **contratos de autorización** están documentados en la especificación de la API (e.g., OpenAPI) indicando qué roles o scopes son necesarios en cada endpoint, sirviendo como garantía formal de seguridad entre el front-end, el gateway y los servicios.

# Monitoreo, Métricas y Alertas\_cc

## 13.1 Métricas de negocio y técnicas:

El contexto ControlCajaBC expone métricas tanto técnicas como de negocio para evaluar la salud del servicio y el uso del módulo de caja. Entre las métricas **técnicas** se incluyen la **latencia** de las operaciones clave (por ejemplo, el tiempo de respuesta al abrir o cerrar caja), la **tasa de éxito** o porcentaje de llamadas sin error (códigos 2XX) vs. fallidas, y conteos de errores por tipo. Las métricas **de negocio** reflejan el comportamiento operacional: la **cantidad de cierres de caja** realizados en un periodo (por día, semana, etc.), la **tasa de cierres con discrepancia** (porcentaje de sesiones cerradas que presentaron diferencia entre lo esperado y lo contado), el **monto promedio de discrepancia** cuando ocurre (para identificar desviaciones inusuales de efectivo), y el número de cajas abiertas simultáneamente. Estas métricas de negocio ayudan a la microempresa a detectar problemas como cuellos de botella en cierres o posibles manejos inadecuados de efectivo. Por ejemplo, una **tasa de discrepancias** superior al umbral esperado podría indicar necesidad de revisar procedimientos de caja o posibles incidencias de fraude.

## 13.2 Herramientas e integración (Prometheus, Grafana):

Factura Fácil integra un stack de monitoreo basado en **Prometheus** y **Grafana**. El microservicio ControlCajaBC expone un endpoint de métricas (ej.: `/metrics`) en formato compatible con Prometheus, incluyendo contadores, histogramas y gauges para las métricas antes mencionadas. Prometheus periódicamente consulta este endpoint y almacena los datos de series temporales. Por ejemplo, métricas como `caja_cierre_latency_seconds_bucket` (histograma de latencia de cierres) o `caja_cierres_total` (contador de cierres realizados) son recolectadas automáticamente. Luego, **Grafana** se utiliza para visualizar estas métricas en dashboards personalizados: se pueden crear paneles con gráficos de latencia (p50, p90, p99 de tiempos de cierre), paneles mostrando el número de cajas cerradas por día, y tablas o alertas sobre discrepancias. Esta integración permite a los administradores de la plataforma y a soporte técnico vigilar en tiempo real el funcionamiento de ControlCajaBC. Adicionalmente, se contempla la integración con un sistema de log centralizado (p.ej., Elastic Stack) para correlacionar métricas con eventos de log si se requiere un análisis más profundo, aunque las métricas de Prometheus cubren la visión general del servicio.

## 13.3 Alertas, SLIs/SLAs sugeridos:

Sobre la base de estas métricas, se han definido **SLIs** (Service Level Indicators) y **SLAs** (Service Level Agreements) internos que guían las alertas. Por ejemplo, un SLI clave es la latencia de cierre de caja: se establece que el 95% de las operaciones de cierre deberían completar en <2 segundos y el 99% en <5 segundos. Consecuentemente, el SLA podría acordar que “la operación de cerrar caja tendrá una disponibilidad de 99.9% y una latencia p95 ≤ 2s en horario comercial”. Si estos objetivos se incumplen, se generan **alertas** automáticas. El sistema de alertas (vía Prometheus Alertmanager u otra herramienta) está configurado, por ejemplo, para enviar una notificación si: la latencia p95 supera 2s por más de 5 minutos, la tasa de errores en operaciones de apertura/cierre excede 1% en la última hora, o si más del X% de cierres diarios presentan discrepancias. Asimismo, una alerta crítica podría dispararse si alguna caja lleva abierta un tiempo excesivo sin cierre (indicando posible olvido). Estos umbrales se ajustan de acuerdo al comportamiento normal esperado de las microempresas usuarias. En términos de acuerdos de nivel de servicio, Factura Fácil podría garantizar a sus clientes empresariales una disponibilidad mensual mínima del módulo de Control de Caja (por ejemplo, SLA de 99.8%), respaldada por las métricas de uptime y tiempo de respuesta monitoreadas. Todas las alertas y SLAs se documentan para la operación, de modo que el equipo de DevOps pueda reaccionar rápidamente ante desviaciones y asegurar la continuidad del servicio.

# Estrategia de Testing\_cc

## 14.1 Pruebas unitarias para agregados y servicios de dominio:

La lógica de negocio del contexto ControlCajaBC se prueba exhaustivamente con **pruebas unitarias** aisladas. Cada *Aggregate Root* (por ejemplo, la entidad CajaSesion) cuenta con una batería de tests para verificar las invariantes y reglas de negocio: casos como impedir cerrar una caja que no ha sido abierta, calcular correctamente la diferencia dado un conjunto de transacciones de venta simuladas, o no permitir aperturas duplicadas para el mismo cajero. Estas pruebas utilizan frameworks como **xUnit** (en entornos .NET) para la estructura de test, junto con bibliotecas de *mocking* como **Moq** para simular dependencias externas (por ejemplo, repositorios o servicios de tiempo). Adicionalmente, los **Domain Services** (servicios de dominio que encapsulan lógica compleja que no cabe en un solo agregado) también son cubiertos por pruebas unitarias. Por ejemplo, si existe un servicio de dominio que calcula el monto esperado de efectivo a partir de las ventas del día (integrando posiblemente varias fuentes), se le crean pruebas con distintos escenarios (ventas normales, devoluciones, descuentos, etc.) para asegurar su correcto funcionamiento.

## 14.2 Pruebas de contrato de integración:

Dado que ControlCajaBC interactúa con otros bounded contexts (p.ej., podría consumir eventos del contexto de Ventas o emitir eventos al contexto Contabilidad), se implementan **pruebas de contrato** para garantizar la compatibilidad entre servicios. Usando herramientas como **Pact**, se definen contratos que describen la interacción esperada. Por ejemplo, si ControlCajaBC publica un evento `CajaCerrada` una vez completado el cierre, los consumidores (como ContabilidadBC) pueden tener tests de pacto que verifiquen que el evento contiene los campos y formatos acordados. Inversamente, si ControlCajaBC depende de un evento `VentaRegistrada` del contexto de Ventas, se define un contrato del lado consumidor que simula dicho evento y comprueba que ControlCajaBC lo procesa adecuadamente (actualizando el total esperado de caja, por ejemplo). Estas pruebas de contrato se ejecutan en los pipelines de integración continua, de modo que cualquier cambio en la API o esquemas de eventos que rompa un contrato establecido haga fallar las pruebas antes de llegar a producción. Esto asegura integraciones robustas y evita sorpresas en tiempo de ejecución.

## 14.3 Pruebas end-to-end de casos de uso:

Más allá de las pruebas aisladas, se realizan **pruebas end-to-end** que cubren escenarios completos de usuario a través de múltiples componentes. En un entorno de staging (o QA) que replica la configuración productiva, se automatiza por ejemplo el siguiente caso: *Apertura de caja → Registro de varias ventas en efectivo (en el sistema de Ventas) → Cierre de caja → Verificación de que el estado final de la sesión y el total de ventas coinciden*. Esta prueba end-to-end involucra la interacción real entre el front-end (o simulación del mismo vía scripts/API), el servicio ControlCajaBC, el servicio de VentasBC y posiblemente otros (Notificaciones, Contabilidad). Se verifica no solo la funcionalidad (que la caja pueda abrirse y cerrarse correctamente) sino también la **integración** (que los eventos o llamadas entre servicios ocurran en el orden y formato esperado). Frecuentemente se apoyan estas pruebas en frameworks como **Selenium** o **Playwright** para automatizar la interfaz de usuario si es web, o simplemente mediante suites de API testing con herramientas como **Postman/Newman** para simular llamadas REST de extremo a extremo. Los casos de uso críticos (apertura, cierre, manejo de discrepancias) cuentan con al menos un test end-to-end automatizado.

## 14.4 Frameworks sugeridos:

Para implementar lo anterior, el proyecto sugiere una serie de frameworks estándar en el stack tecnológico utilizado:

- **xUnit:** Framework de unit testing para .NET, utilizado para escribir pruebas unitarias legibles y categorizarlas por componente. Por ejemplo, pruebas de la lógica del agregado CajaSesion.

- **Moq:** Biblioteca de *mocking* para .NET que permite crear dobles de prueba (mocks/stubs) de interfaces y clases. Se usa en pruebas unitarias para simular repositorios (p.ej., simular que la base de datos guarda o lanza excepción) y otros servicios durante el test.
- **Pact:** Framework para pruebas de contrato consumer-producer. Pact facilita definir, en tests automatizados, el contrato esperado entre un consumidor (p.ej., ControlCajaBC consumiendo un API de otro contexto) y un proveedor. Se integra en CI para verificar que las respuestas reales del proveedor cumplen el contrato pactado.
- **FluentAssertions:** (Opcional) Librería para .NET que ofrece una sintaxis expresiva para validar resultados en pruebas, mejorando la legibilidad de los tests.
- **TestContainers:** (Opcional) Para pruebas de integración de base de datos, se pueden usar contenedores efímeros de SQL Server o la base correspondiente, levantados durante el test, garantizando un entorno limpio y reproducible.

Con esta combinación de pruebas en múltiples niveles, el equipo asegura que el **Bounded Context ControlCajaBC** se comporta correctamente aislado y en conjunto con el resto del sistema Factura Fácil, detectando regresiones de manera temprana durante el ciclo de desarrollo.

# Decisiones de Arquitectura (ADR)\_cc

## 15.1 Registro de decisiones:

A lo largo del diseño e implementación de ControlCajaBC se han documentado varias **decisiones arquitectónicas clave (ADR)**, junto con su motivación y consecuencias. A continuación se resumen las más relevantes:

- **Integración Event Bus vs REST:**

- *Decisión:* Se optó por un modelo de integración asíncrono mediante un **Event Bus** para la comunicación entre bounded contexts, en lugar de invocaciones REST sincrónicas punto a punto. *Motivación:* El bus de eventos (por ejemplo, usando RabbitMQ) reduce el acoplamiento temporal entre servicios – VentasBC publica eventos de venta, y ControlCajaBC los consume cuando está disponible, permitiendo resiliencia y **consistencia eventual** en el cálculo del efectivo. También soporta mejor la escalabilidad, ya que múltiples servicios pueden suscribirse a los eventos sin que el emisor tenga conocimiento directo. REST quedó reservado para consultas sincrónicas ligeras o casos donde realmente se necesita respuesta inmediata de otro servicio (lo cual es mínimo en este contexto). Esta decisión mejora la tolerancia a fallos (si ControlCajaBC está temporalmente caído, procesa las ventas pendientes al recuperarse) a cambio de una mayor complejidad en la infraestructura de mensajería.

- **Base de Datos Relacional vs NoSQL:**

- *Decisión:* Se eligió usar una **Base de Datos Relacional** (ej. PostgreSQL/SQL Server) para persistencia de las sesiones de caja, en lugar de una base NoSQL. *Motivación:* Los datos de caja son altamente estructurados y requieren **consistencia transaccional** fuerte (p.ej., registrar apertura y cierre de forma atómica, cálculos precisos de diferencias). Un RDBMS ofrece integridad referencial y soporte nativo a transacciones ACID, crucial para evitar pérdidas o inconsistencias de datos financieros. Si bien una solución NoSQL podría ofrecer flexibilidad en el esquema o escalabilidad horizontal sencilla, en el caso de microempresas el volumen de datos de caja no es tan grande y un solo nodo relacional puede manejarlo con solvencia. Adicionalmente, el equipo y las herramientas existentes están más familiarizados con SQL, reduciendo la curva de aprendizaje y permitiendo aprovechar ORM (Mapeo Objeto-Relacional) para agilizar el desarrollo. Se consideró NoSQL para almacenar eventos de dominio o logs, pero finalmente se mantuvo la persistencia principal en relacional por las razones expuestas.

- **Patrones de integración (Coreografía vs Orquestación):**

- *Decisión:* Se implementó un patrón de integración **coreografiado por eventos** en lugar de una orquestación centralizada de workflows. *Motivación:* En la coreografía, cada microservicio reacciona a eventos de dominio relevantes: p. ej., VentasBC emite `VentaRegistrada` y ControlCajaBC actualiza su estado; luego ControlCajaBC emite `CajaCerrada` y ContabilidadBC genera el asiento contable correspondiente. No existe un orquestador central dictando el flujo, lo que simplifica la arquitectura y elimina puntos únicos de fallo. Este patrón encaja bien dado que las acciones de caja (abrir, cerrar) son principalmente desencadenadas por eventos de usuario locales y por la finalización del día de ventas, sin requerir una secuencia compleja multi-servicio que amerite un saga central. La desventaja es que el flujo completo de procesos distribuidos (como “día de ventas completado”) está implícito en los mensajes,

requiriendo buena documentación y seguimiento de eventos, pero se consideró manejable en comparación con introducir un orquestador dedicado.

- **Herramientas de mensajería y orquestación:**

- *Decisión:* Se adoptó **RabbitMQ** como el bus de mensajería para publicar y consumir eventos entre contextos, apoyado por la librería **MassTransit** en los microservicios .NET para simplificar la integración.  
*Motivación:* RabbitMQ es una solución probada y robusta que soporta patrones de *pub/sub*, aseguramiento de entrega (ACK/retry) y escalamiento apropiado para los volúmenes de eventos esperados en Factura Fácil. MassTransit provee un nivel de abstracción que permite definir consumers de eventos y end-points con mínima configuración, agilizando el desarrollo. En cuanto a **orquestación**, tras evaluar opciones (como usar un motor de workflow tipo **Durable Functions** de Azure, o Saga patterns con frameworks específicos), se decidió que no era necesario incorporar una herramienta adicional. La combinación de eventos de dominio y consistencia eventual cubre las necesidades sin agregar complejidad. Cada decisión ADR fue documentada en el repositorio del proyecto, de modo que en el futuro se pueda revisar su contexto y, si es necesario, revertir o ajustar con base en nuevas circunstancias.

# CatalogoArticulosBC -Antonio H

## 1.1 Propósito

**CatalogoArticulosBC** gestiona de forma centralizada todo el catálogo de productos y servicios de *Factura Fácil*, garantizando datos coherentes, completos y alineados con requerimientos fiscales, contables y de negocio. Sus principales objetivos incluyen:

- **Mantener el catálogo de ítems:** Definir y administrar productos **simples, variantes y combos** con todos sus atributos relevantes (descripciones, códigos, categorías, etc.).
- **Cumplimiento tributario:** Configurar atributos fiscales (afectación de IGV, códigos SUNAT, unidades de medida) para cada producto, evitando rechazos de comprobantes electrónicos por datos incorrectos.
- **Integración contable:** Asociar cuentas contables, centros de costo y presupuestos a los ítems para automatizar la contabilización y análisis financiero.
- **Contenido multimedia:** Adjuntar imágenes y documentos técnicos (manuales, fichas) a los productos, enriqueciendo la información disponible para el usuario.
- **Escalabilidad del catálogo:** Permitir la **importación y exportación masiva** de productos (archivos Excel/CSV) para agilizar la carga inicial y migraciones desde otros sistemas.
- **Servicios de consulta:** Proveer servicios a otros módulos (facturación, precios, etc.) para **validar SKU** y obtener atributos de productos de forma consistente.

En síntesis, **CatalogoArticulosBC** asegura que toda la información de productos/servicios sea única, válida y accesible para los demás procesos de *Factura Fácil*.

## 1.2 Responsabilidades Clave

Las principales responsabilidades (funcionalidades) que asume *CatalogoArticulosBC* se listan a continuación:

Responsabilidad	Descripción
<b>Crear Producto Simple</b>	Registrar un ítem básico con SKU único, incluyendo sus atributos fiscales (IGV, código SUNAT), contables (cuenta, centro de costo) y de inventario.
<b>Crear Producto Variante</b>	Generar una variante de un producto existente, heredando atributos comunes del producto base pero añadiendo atributos diferenciadores (ej. talla, color).
<b>Crear Producto Combo</b>	Definir un producto <i>combo</i> que agrupa múltiples ítems (simples o variantes) bajo un SKU de paquete, especificando componentes, cantidades y un precio total de combo.

<b>Editar Ítem</b>	Modificar los atributos de un producto existente, por ejemplo descripción, categoría, afectación IGV, datos contables, peso, etc..
<b>Deshabilitar Ítem</b>	Inactivar un ítem del catálogo (soft-delete), bloqueando su uso en operaciones futuras (no aparecerá en facturación ni en listas de precios).
<b>Gestionar Multimedia</b>	Adjuntar, actualizar o eliminar archivos multimedia asociados al producto (imágenes, manuales, PDFs), manteniendo un repositorio de recursos relacionados.
<b>Importar Productos (Masivo)</b>	Procesar archivos <b>Excel/CSV</b> para cargar en lote nuevos productos o actualizar existentes, con validaciones de formato y datos (ej. SKU únicos).
<b>Exportar Productos</b>	Generar un archivo <b>Excel/CSV</b> con el catálogo completo o filtrado, facilitando respaldos o ediciones masivas externas. <i>(Esta funcionalidad es complementaria a la importación masiva.)</i>
<b>Consultar Catálogo</b>	Buscar y filtrar ítems por diferentes criterios (SKU, nombre, categoría, estado, unidad de medida, etc.), proporcionando vistas del catálogo para la gestión y selección de productos.
<b>Validar SKU</b>	Proveer un servicio (consulta) para validar la existencia y obtener detalles de un producto dado su SKU, utilizado por módulos externos (facturación, precios) antes de registrar transacciones.

### 1.3 Contexto y Motivación ☀

En las microempresas es crítico contar con un catálogo de productos bien definido y **único**. *Factura Fácil* requiere información precisa de cada ítem en las facturas electrónicas para evitar rechazos por parte de la autoridad tributaria (SUNAT) y discrepancias contables. Además:

- Las PYMEs necesitan manejar **variantes** de productos (ej. tallas, colores) y crear **combos o paquetes promocionales** sin duplicar datos, de manera sencilla.
- Se demanda **uniformidad** en los atributos tributarios (IGV, código de producto SUNAT) y contables para integrarse correctamente con los procesos de facturación y cuentas por cobrar.

- La posibilidad de adjuntar **imágenes y documentos** por producto enriquece la información disponible, mejorando la experiencia de usuario al identificar artículos y brindando detalle técnico cuando sea necesario.
- La **importación masiva** de productos desde Excel u otras fuentes acelera la carga inicial del catálogo o la migración desde sistemas existentes, reduciendo trabajo manual y errores.

En resumen, este contexto motiva la existencia de *CatalogoArticulosBC*: garantizar calidad y consistencia en datos de productos para soportar eficientemente la emisión de comprobantes, la gestión de precios y el análisis de ventas.

## 1.4 Fuentes de Información

El diseño de *CatalogoArticulosBC* se apoya en diversas fuentes de información y normativas:

- **Normativas tributarias (SUNAT):** Catálogos oficiales de SUNAT, como el catálogo de **códigos de productos/servicios** (Código SUNAT de 8 dígitos requerido en facturas) y el catálogo de **unidades de medida** permitidas (NIU, KG, etc.), asegurando que los datos de producto cumplan estándares legales.
- **Normas contables locales:** El **Plan Contable General Empresarial (PCGE)** y políticas contables de la empresa informan los campos contables (cuentas contables, centros de costo, presupuestos) asociados a cada producto, para que la integración con contabilidad sea correcta.
- **Stakeholders y requisitos de negocio:** Aportes de **usuarios clave** (dueños de negocio, contadores, vendedores) identificaron la necesidad de variantes y combos, importación masiva, y gestión multimedia, moldeando los **casos de uso** del catálogo.
- **Sistemas internos existentes:** Los módulos de **Configuración del Sistema** (para catálogos maestros de unidades, impuestos, etc.) e **Identidad & Autenticación** (para manejo de multi-tenant y roles de usuario) proveen lineamientos para que el catálogo se integre en la plataforma SaaS de manera consistente. Por ejemplo, *CatalogoArticulosBC* utiliza servicios del *ConfiguracionSistemaBC* para obtener listas actualizadas de unidades de medida e impuestos permitidos.
- **Migración de datos:** Estructuras de datos de sistemas previos (hojas de cálculo de productos que usan los clientes, otros ERPs sencillos) fueron analizadas para definir formatos de importación/exportación y asegurar que la herramienta pueda incorporar fácilmente catálogos existentes.

Estas fuentes han alimentado el diseño de *CatalogoArticulosBC*, garantizando que responda a la realidad normativa peruana y a las necesidades prácticas de las microempresas usuarias de *Factura Fácil*.

## 1.5 Modelo de Dominio

*CatalogoArticulosBC* organiza su modelo de dominio en **tres agregados raíz**, según el tipo de ítem gestionado: **ProductoSimple**, **ProductoVariante** y **ProductoCombo**. Cada uno representa un tipo de producto distinto, con atributos y comportamientos particulares. Además, se definen **objetos de valor** para encapsular ciertos atributos y **servicios de dominio** para lógica específica que trasciende las entidades individuales.

### 1.5.1 Agregados Raíz y Entidades Principales:

- **ProductoSimple** – *Producto individual básico*. Es el agregado principal del catálogo; posee un **SKU único** que lo identifica globalmente. Contiene atributos descriptivos (nombre, descripción, categoría), atributos **tributarios** (unidad de medida, afectación IGV, código SUNAT), atributos **contables** (cuenta contable, centro de costo, presupuesto asignado) y opcionalmente atributos de **inventario** (peso, dimensiones, etc.). Un **ProductoSimple** puede tener **múltiples variantes** asociadas (ej. una prenda con distintas tallas) y puede ser componente de uno o varios combos. Mantiene métodos para validar la unicidad de su SKU, actualizar sus atributos y **deshabilitarse** (cambio de estado a inactivo).
- **ProductoVariante** – *Variante de un ProductoSimple existente*. Representa una versión específica de un producto base, diferenciada por ciertos atributos particulares (por ejemplo, color o talla de una prenda). La variante **hereda**

los atributos comunes de su producto padre, pero añade sus propios valores en los atributos diferenciadores. En el modelo, mantiene una referencia **N:1 al ProductoSimple** correspondiente. Tiene entidades internas como **AtributoVariante** (pares nombre-valor que definen cada característica diferenciadora). Sus invariantes incluyen que el **producto padre debe permanecer activo** para que la variante sea válida (no se pueden usar variantes de un producto deshabilitado). Expone comportamientos para validar combinaciones de atributos únicos por variante (p.ej., que no existan dos variantes con la misma talla y color), actualizar sus datos propios y deshabilitarse.

- **ProductoCombo** – *Producto compuesto o paquete.* Es un agregado que **agrupa múltiples productos** (ya sean simples o variantes) bajo un nuevo SKU, representando un paquete promocional o kit. Almacena la lista de **componentes** del combo, cada uno con referencia al producto incluido y la cantidad correspondiente (entidad **ComponenteCombo**). Un combo puede, por ejemplo, consistir en “1 teclado + 1 mouse” como un solo producto paquete. Sus invariantes dictan que **no puede incluir ítems inactivos ni otros combos** como componentes. Además, suele registrar un **peso total** (suma de pesos de componentes) para efectos logísticos. Provee métodos para validar que todos sus componentes están activos al armar el combo, para calcular/actualizar su peso total automáticamente según componentes, y para deshabilitar el combo completo.

- **Otras Entidades (del Modelo Interno):**

- **AtributoVariante** – Objeto inmutable que encapsula una característica específica de una variante (nombre de atributo y valor, e.g. “Talla: M”). No tiene identidad propia (se identifica por sus valores) y pertenece al contexto de un ProductoVariante. Sirve para representar ordenadamente propiedades extensibles de variantes.
- **ComponenteCombo** – Entidad perteneciente a un ProductoCombo que representa cada **línea de componente** dentro de un combo. Contiene la referencia (ID) al producto incluido y la cantidad/unidades de ese producto en el combo. Puede incluir una regla para validar cantidades mínimas/máximas. No posee vida fuera del combo (su identidad está compuesta por el combo y el producto referenciado).
- **MultimediaProducto** – Representa un **recurso multimedia** asociado a un producto (sea simple, variante o combo). Puede almacenar metadatos del archivo (nombre, tipo MIME, URL o almacenamiento) y métodos para validar el formato o tamaño del archivo y para eliminar/adjuntar el recurso. Esta entidad permite adjuntar múltiples imágenes u otros documentos a un ítem sin afectar sus datos principales.

#### 1.5.2 Objetos de Valor (Value Objects):

Los principales objetos de valor definen atributos clave de los productos, siendo inmutables y comparables por su contenido. Algunos de ellos provienen de catálogos estándar del sistema o externos:

- **UnidadMedida** – Representa la unidad de medida comercial del ítem (p.ej. NIU – unidad, KG – kilogramo, LT – litro, etc.). Se modela como un enum o catálogo inmutable, restringido a los códigos permitidos por SUNAT. La unidad de medida es obligatoria para emitir comprobantes y guías de remisión, por tanto el sistema valida que el valor asignado al producto pertenezca al catálogo vigente.
- **AfectacionIGV** – Indica la categoría tributaria respecto al IGV (Impuesto General a las Ventas) que aplica al producto: **GRAVADO**, **EXONERADO**, **INAFECTO** o **SUJETO A DETRACCIÓN**, entre otros. Es un objeto valor enumerado según las definiciones tributarias oficiales. Este valor determina el cálculo de impuestos en la facturación.
- **CodigoSUNAT** – Código numérico de 8 dígitos que clasifica el producto/servicio según el catálogo oficial de SUNAT. Es obligatorio incluir este código en el XML de los comprobantes electrónicos (UBL) para ciertos tipos de ítems. El sistema verifica que el código ingresado exista en el catálogo oficial vigente (descargado o referenciado en Configuración del Sistema).

- **CuentaContable** – Código de la cuenta contable asociada a los ingresos de la venta de este ítem (por ejemplo “70111 – Ventas mercadería nacional”). Es una cadena alfanumérica siguiendo el plan contable interno de la empresa. Este VO asegura que solo se usen cuentas válidas y facilita la integración con módulos contables (al momento de generar asientos, se usa la cuenta del producto vendido).
- **CentroCosto** – Identificador de centro de costo para imputar las ventas de este producto (ejemplo: “CC-01” para Ventas tienda Lima). Se utiliza para clasificar internamente las operaciones y sacar reportes financieros segmentados.
- **Presupuesto** – Monto presupuestado o estimado relacionado al producto (por ejemplo, el costo objetivo o meta de ventas). Es un valor decimal posiblemente con una moneda asociada. Se usa para planificar y luego comparar con ventas reales. En el catálogo, este VO puede ser opcional y sirve de referencia para análisis financieros.
- **Peso** – Peso físico bruto del producto en kilogramos. Es relevante principalmente para productos físicos que puedan figurar en **guías de remisión** o para calcular costos de envío. Si un producto es intangible (servicio), podría ser nulo o cero. Se exige que si el producto es de tipo físico/tangible, el peso sea > 0 (regla RN-CA-003).

Nota: Otros valores como **SKU** en sí mismo podrían tratarse como VO (por ejemplo, para encapsular un formato específico), pero en este contexto se maneja como atributo String con restricción de unicidad.

### 1.5.3 Servicios de Dominio:

Algunas lógicas complejas o transversales en *CatalogoArticulosBC* se implementan en **Servicios de Dominio**, ya que no pertenecen exclusivamente a una entidad. Estos servicios encapsulan reglas de negocio que involucran múltiples agregados o interacción con otros contextos:

- **ServicioValidacionSKU**: Coordina la lógica de **validación de SKU** cuando un módulo externo (p.ej. el Bounded Context de Comprobantes Electrónicos) solicita verificar un código. Este servicio puede comprobar que el SKU exista y, de estar **inhabilitado**, rechazar su uso. También retorna detalles básicos del producto (unidad, afectación IGV, descripción, peso) necesarios para emitir un comprobante. En su implementación podría consultar repositorios internos y aplicar reglas de negocio (como que el producto esté activo).
- **ServicioImportacionCatalogo**: Orquesta la **lectura y procesamiento de archivos Excel/CSV** para alta masiva. Este servicio de dominio se encarga de parsear el archivo, validar cada fila (formato de datos, unicidad de SKU, existencia de códigos válidos) y luego invocar la creación o actualización de productos correspondientes. Maneja transacciones o unidades de trabajo en lote para garantizar consistencia (p.ej., si alguna fila falla, revertir o reportar adecuadamente).
- **ServicioMultimediaProducto**: Encapsula reglas para la gestión de archivos asociados. Por ejemplo, puede verificar **formatos permitidos** (imágenes JPG/PNG hasta cierto tamaño, PDF para manuales, etc.) y estandarizar la manera en que los archivos se guardan (integración con un *File Storage* o base de datos). Aunque gran parte de su implementación podría estar en la infraestructura, desde el dominio se define que, por ejemplo, solo ciertos tipos de archivos son aceptados para un producto (política de negocio).
- **Servicios Externos Integrados**: *CatalogoArticulosBC* interactúa con servicios de otros dominios que, si bien no son implementados en este contexto, definen **contratos** importantes: por ejemplo, un **API Service de Facturación** externo que podría requerir datos de productos para generar el XML de factura (precio unitario, peso para guía, descripción conforme a SUNAT) o el servicio de **Identidad y Autenticación** (dominio “Mi Contasis”) que proporciona información de tenant y usuario para autorizar operaciones. Estos servicios no son lógicos del catálogo en sí, pero el catálogo define cómo se consumen: p.ej., a través de interfaces como `ObtenerCatalogos()` del *ConfiguracionSistemaBC* para traer listas actualizadas de unidades, impuestos, etc.. También se podrían exponer métodos como `CatalogoArticulosBC.validarSku(sku)` que otros contextos consumen vía mensajes o REST, actuando el *CatalogoArticulosBC* como *supplier* de información de productos. (Ver **Relaciones con otros BC** abajo).

En todos los casos, los **Servicios de Dominio** aseguran que las **reglas de negocio** se apliquen consistentemente, ya sea que la operación se dispare desde una acción del usuario o desde otro contexto.

## 1.6 Relaciones con otros Bounded Contexts ↗

CatalogoArticulosBC no opera aislado; interactúa con varios otros **Bounded Contexts (BC)** dentro del ecosistema *Factura Fácil* para proveer y consumir información:

- **ConfiguracionSistemaBC** (Dominio de Configuración): CatalogoArticulosBC **consume** de este BC los catálogos maestros necesarios para validar sus datos. Por ejemplo, obtiene la lista vigente de **Unidades de Medida**, las categorías de **Afectación IGV**, o el plan de **Cuentas Contables y Centros de Costo**. Esto suele realizarse vía consultas sincrónicas (`ObtenerCatálogos()`) o suscripción a eventos de actualización de dichos catálogos. Esta relación asegura que el catálogo de artículos esté alineado con la configuración global del sistema (e.g., si se añade una nueva unidad de medida en Configuración, se refleje como opción válida en los productos).
- **ComprobantesElectonicosBC** (Dominio de Facturación): Este contexto de facturación **consume datos de productos** para validar líneas de comprobantes. Al momento de agregar un ítem a una factura/boleta, ComprobantesElectonicosBC invoca a CatalogoArticulosBC (por mensaje o API) para **validar el SKU** ingresado y obtener sus atributos clave: descripción, unidad de medida, categoría IGV y peso (si es relevante para guías de remisión). Así se evitan errores en la emisión (por ejemplo, no facturar un producto inexistente o inactivo). Además, CatalogoArticulosBC publica un evento **ProductoInhabilitado** cuando un ítem se desactiva, que ComprobantesElectonicosBC escucha para en adelante **bloquear** el uso de ese producto en nuevas facturas. (Esta integración sigue un patrón **Publisher-Subscriber**, donde Comprobantes actúa como suscriptor de ciertos eventos del Catálogo.)
- **ListaPreciosBC** (Dominio de Precios): Maneja las listas y reglas de precios de venta. Este contexto depende del catálogo para asociar precios a productos válidos. CatalogoArticulosBC publica eventos **ProductoCreado** y **ProductoModificado**, que ListaPreciosBC consume para **actualizar sus listas de precios** automáticamente. Por ejemplo, al crearse un nuevo producto, ListaPreciosBC podría generar un registro de precio base por defecto; o si se modificó la categoría de IGV de un producto, ListaPreciosBC recalcula precios con impuestos. Además, cuando ListaPreciosBC necesita calcular un precio en tiempo real (p.ej. en la pantalla de factura), puede requerir datos del producto (SKU, unidad, afectación) que obtiene del catálogo. Esta relación es de tipo **Customer-Supplier**, donde ListaPreciosBC es cliente que confía en CatalogoArticulosBC como fuente única de verdad para datos de productos.
- **IndicadoresNegocioBC** (Dominio de Reportes/BI): Este contexto genera métricas e informes de ventas. Si bien IndicadoresNegocioBC principalmente recibe eventos de ventas (emitidos por Comprobantes o ControlCaja), utiliza la información del catálogo para enriquecer sus reportes (por ejemplo, nombre del producto, categoría). CatalogoArticulosBC publica eventos como **ProductoCreado/Modificado**, pero IndicadoresNegocioBC podría no suscribirse directamente a ellos (no crítico para métricas). Sin embargo, IndicadoresNegocioBC sí puede recibir indirectamente eventos de **venta por producto** (cada vez que se vende un SKU, proveniente de Comprobantes) y luego consultar al catálogo para obtener detalles adicionales de ese SKU para el dashboard. En el Context Map se considera que IndicadoresNegocioBC obtiene **lectura de ventas por SKU** y genera rankings; para ello requiere la existencia de los SKUs en el catálogo. Esta interacción se puede implementar vía consultas eventuales (un servicio de consulta de producto desde Indicadores) o mediante proyecciones (replicar algunos datos de productos en el contexto de indicadores). El vínculo es **indirecto**, pero importante: el catálogo asegura que los SKU referidos en las ventas correspondan a datos válidos.
- **GestionClientesBC** (Dominio de Clientes): Entre Catálogo de Artículos y Gestión de Clientes **no hay una relación directa fuerte** en cuanto a datos de productos. Sin embargo, en flujos de negocio completos (p. ej., una venta), los datos de cliente y de producto confluyen en la factura. En nuestro diseño, GestionClientesBC provee información

de clientes (como límites de crédito, preferencias) a otros contextos como Facturación o Caja, pero no interactúa directamente con productos. Podría haber interacciones indirectas, por ejemplo, una promoción aplicada a un cliente específico podría involucrar un producto, pero eso sería mediado por *ListaPreciosBC* o *ComprobantesElectonicosBC*. Por lo tanto, se considera **sin relación directa** con *CatalogoArticulosBC*, más allá del uso común dentro del flujo de facturación.

- **ControlCajaBC** (Dominio de Caja): Similar a clientes, el contexto de Control de Caja (manejo de turnos, ingresos/egresos) no requiere datos específicos del catálogo de artículos. *ControlCajaBC* se encarga de la apertura/cierre de caja y movimientos de dinero, y asocia los movimientos con comprobantes emitidos, pero la información de qué productos se vendieron en cada comprobante no es relevante para las operaciones de caja (solo el total monetario). Por ende, no hay contratos directos ni eventos compartidos entre *ControlCajaBC* y *CatalogoArticulosBC* en este diseño. Cada uno maneja un aspecto distinto de la solución, coordinados a través del documento de venta (factura/boleta) en común pero sin comunicarse directamente.
- **Identidad y Autenticación (Mi Contasis)**: Este dominio transversal gestiona usuarios, roles y **tenancy** (multi-empresa). Aunque no intercambia eventos de negocio con *CatalogoArticulosBC*, sí impone **contexto de seguridad**: el catálogo es *multi-tenant*, por lo que cada consulta o modificación de productos está asociada a una empresa/tenant específica. El *Orquestador de Apps* (un gateway u orquestador global) se asegura de incluir el identificador del tenant en las solicitudes hacia *CatalogoArticulosBC*, y este valida los permisos (ej. solo un usuario con rol *Administrador* o *Almacenero* puede crear/editar productos). Ver la sección de **Seguridad (12)** para más detalles. Así, la relación aquí es a través de infraestructura de autenticación/autorización, no de mensajes de dominio.

En resumen, *CatalogoArticulosBC* actúa como **Supplier** de información de productos para los contextos de Facturación y Precios, y como **Consumer** de catálogos maestros desde Configuración del Sistema. Estas interacciones están cuidadosamente definidas mediante interfaces (servicios de consulta sincrónica) y eventos de dominio publicados, garantizando un **acoplamiento bajo** y consistencia de la información de productos en toda la solución.

## 1.7 Restricciones y Reglas del Negocio

Para mantener la integridad del catálogo y cumplir las políticas corporativas y fiscales, *CatalogoArticulosBC* establece varias **invariantes y reglas de negocio** que siempre deben respetarse:

- **Restricción (Invariante) – SKU único:** Ningún otro producto (simple, variante o combo) puede tener el mismo código SKU. El SKU funciona como identificador único a nivel de todo el catálogo. *Implicación:* Antes de crear un producto nuevo o importar una lista, el sistema verifica que el SKU no exista ya; en caso contrario, rechaza la operación.
- **Restricción – Variante requiere padre activo:** Una **ProductoVariante** solo es válida si su producto padre (**ProductoSimple**) está en estado ACTIVO. *Implicación:* Si se intenta usar una variante cuyo padre fue deshabilitado, se debe bloquear (no permitir ventas). Además, el sistema podría inhabilitar automáticamente todas las variantes cuando se deshabilita el producto base.
- **Restricción – Combo no incluye ítems inválidos:** Un **ProductoCombo** no puede contener en su lista de componentes ni productos inactivos ni otros combos. Es decir, solo puede componerse de productos simples o variantes que estén activos. *Implicación:* Al armar o editar un combo, el servicio valida cada ID de producto agregado; si alguno está inactivo, la operación es rechazada. Tampoco se permiten combos anidados (un combo dentro de otro).

- **Regla RN-CA-001 – Código SUNAT válido:** El campo **CodigoSUNAT** de un producto debe corresponder a una entrada existente en el catálogo oficial de bienes y servicios de SUNAT. *Implicación:* Al guardar un producto, el sistema puede verificar contra una tabla actualizada de códigos SUNAT; un código inválido (o vacío cuando debería tener valor) provocará un error de validación. Esto asegura el cumplimiento con la emisión electrónica (que requiere códigos válidos en el XML).
- **Regla RN-CA-002 – Unidad de medida permitida:** La **UnidadMedida** asignada al producto debe pertenecer al catálogo vigente de unidades soportadas (normalmente definido por SUNAT y configurado en el sistema). *Implicación:* Solo se pueden seleccionar unidades desde la lista proporcionada por *ConfiguracionSistemaBC*. Si una unidad se deja de usar (p.ej. SUNAT la elimina), ya no podrá usarse en nuevos productos.
- **Regla RN-CA-003 – Peso positivo para físicos:** Si un producto es tangible/físico, su atributo **Peso** debe ser mayor a 0. *Implicación:* No se permiten pesos cero o negativos en productos que tengan tipo de unidad de medida física (Kg, lb, etc.), evitando inconsistencias en guías de remisión o cálculos de envío. Productos intangibles pueden llevar peso nulo o se omite el campo.
- **Regla RN-CA-004 – Afectación IGV válida:** El atributo **AfectacionIGV** debe ser uno de los valores permitidos por la normativa (GRAVADO, EXONERADO, INAFECTO, etc.). *Implicación:* El sistema solo acepta esos valores en el campo, garantizando que cada producto esté categorizado correctamente para el cálculo/imposición de impuestos.
- **Regla RN-CA-005 – Eliminación lógica (Soft-delete):** Cuando se “elimina” un producto, en realidad se marca como **Inactivo** en lugar de borrarlo físicamente. Esto aplica especialmente si el producto tiene historial de uso (ventas registradas, etc.). *Implicación:* La regla asegura trazabilidad: los comprobantes antiguos que referencian el producto no pierden la referencia. Al mismo tiempo, productos inactivos ya no aparecen en listados operativos.

Estas restricciones se implementan en las entidades (mediante invariantes en los agregados) y en servicios de dominio (mediante validaciones antes de operaciones). De este modo, *CatalogoArticulosBC* mantiene la **consistencia interna** y evita situaciones que podrían generar errores en otros procesos (ej. facturar un producto duplicado o con datos fiscales inválidos).

## 1.8 Valor Estratégico

El Bounded Context *CatalogoArticulosBC* aporta un **alto valor estratégico** a la solución *Factura Fácil* y al negocio del cliente, ya que:

- **Calidad de datos garantizada:** Centraliza la gestión de productos asegurando información exacta y completa, lo que **evita rechazos de comprobantes por SUNAT** y reduce errores en facturación. Esto genera confianza en el usuario de que sus facturas estarán correctas.
- **Flexibilidad comercial:** Habilita fácilmente **estrategias de venta** como ofrecer variantes (p.ej. distintos modelos de un artículo) y combos promocionales **sin duplicar datos** ni esfuerzos adicionales. Esto puede traducirse en más ventas y promociones rápidas, apoyando al crecimiento del negocio.
- **Integración contable automática:** Al incorporar tributos y cuentas contables en cada producto, *CatalogoArticulosBC* **optimiza la contabilización** de las ventas. Cada vez que se vende algo, ya se sabe a qué cuenta irá esa venta, facilitando reportes financieros y contabilidad sin trabajo manual extra.
- **Eficiencia en puesta en marcha:** Con herramientas de **importación masiva** y administración de multimedia, el catálogo permite **acelerar la implementación** para un nuevo cliente o una nueva tienda. Cargar cientos de productos desde Excel, con imágenes incluidas, puede hacerse en minutos, reduciendo la curva de adopción de la plataforma.
- **Mejora del análisis de negocio:** Al ser la fuente de los datos de SKU usados en ventas, *CatalogoArticulosBC* potencia los dashboards e **informes de Indicadores de Negocio**, proporcionando métricas precisas de ventas por

producto, categorías más vendidas, etc., en tiempo real. Esto ayuda a la toma de decisiones rápidas e informadas (qué productos promover, gestionar stock, etc.).

En suma, *CatalogoArticulosBC* contribuye directamente a los objetivos globales de *Factura Fácil*: rapidez, confiabilidad y simplicidad en la facturación electrónica para microempresas, al brindar un **catálogo sólido** que alimenta todos los procesos posteriores.

# Aggregates, Entities, VO, Domains Events, Policies & Specifications\_ca

## 2.1 Detalle de Elementos del Modelo de Dominio

A continuación, se detalla cada tipo de componente del dominio **CatalogoArticulosBC** – agregados, entidades, objetos de valor, servicios de dominio y eventos – incluyendo sus atributos principales, relaciones, y cómo soportan las reglas/políticas definidas:

### 2.1.1 Agregados (Aggregates)

**ProductoSimple (Agregado raíz):** Representa un producto o servicio individual estándar.

- *Atributos principales:*
  - `productoId` (identidad única interna del agregado)
  - `sku` (código único visible al usuario)
  - `nombre`
  - `descripcion`
  - `unidadMedida` (VO que indica cómo se cuantifica, e.g. NIU, KG)
  - `afectacionIGV` (VO categoría tributaria)
  - `codigoSUNAT` (VO código SUNAT de 8 dígitos)
  - `cuentaContable` (VO cuenta de ingresos asociada)
  - `centroCosto` (VO)
  - `presupuesto` (VO)
  - `peso` (VO, valor decimal).
  - También podría incluir flags como `activo` (estado).
  -
- **Invariantes y reglas:** Debe tener **SKU único** (ver RN-CA-001), `nombre` obligatorio, `unidadMedida` válida y `afectacionIGV` válida. Si es un bien físico, `peso > 0`. Además, su estado define el de sus componentes: no puede deshabilitarse si tiene variantes activas sin inhabilitar (regla derivada).
- **Comportamiento y ciclo de vida:** Comienza con un **Caso de uso Crear Producto Simple**, se le asigna un SKU no usado. Puede recibir actualizaciones mediante **Editar Ítem**, donde valida que cambios cumplan reglas (p.ej., no cambiar a una `unidadMedida` inválida). Al **Deshabilitar Ítem**, cambia su estado a inactivo (soft-delete), emite evento de `ProductoInhabilitado`. Durante su vida, puede agregársele **MultimediaProducto** (adjuntar archivos) o asociarse a listas de precios externamente. Un `ProductoSimple` puede tener **múltiples variantes** (1:N relación) y pertenecer como componente a combos (N:M a través de `ComponenteCombo`, pero la relación se maneja del lado del combo).

**ProductoVariante (Agregado raíz, dependiente de ProductoSimple):** Una variante especializada de un producto base.

- *Atributos:* `productoVarianteId` (ID único), referencia a `productoPadreId` (identificador del `ProductoSimple` del cual deriva), conjunto de **AtributoVariante** (cada uno con nombre-valor describiendo la diferencia, e.g. color=rojo, tamaño=XL), posiblemente propios campos como un SKU derivado o sufijo (aunque

normalmente comparten SKU base con un diferenciador). Hereda implícitamente unidad, código SUNAT, etc., del padre salvo que alguno pueda ser distinto (en general no).

- **Invariante:** Debe existir un ProductoSimple padre **activo** asociado. La combinación de valores de AtributoVariante debe ser **única** entre las variantes de ese mismo producto padre (no pueden haber dos variantes con exactamente los mismos atributos diferenciadores). Cumple también con SKU único global: si se implementa como SKU compuesto (e.g. SKU-base + código variante), igualmente no se repite.
- **Comportamiento:** Se crea vía **Caso de uso Crear Producto Variante**, recibiendo el producto padre y los atributos diferenciadores. Antes de persistir, invoca a *productoPadre* para verificar que esté activo y que la variante propuesta no duplique a otra (posiblemente usando una Specification o comparando valores). Al editar una variante (Ej. cambiar un atributo), se revalida la unicidad de la combinación. Al deshabilitarla, simplemente se marca inactiva; opcionalmente, si el padre se deshabilita, un **Policy** en el dominio podría automáticamente deshabilitar todas sus variantes (o impedir ventas de ellas). Como entidad, *ProductoVariante* suele delegar ciertas lógicas al padre (p.ej., si se requiere algún dato tributario se consulta al padre).

**ProductoCombo (Agregado raíz):** Un producto que es un conjunto de otros productos.

- **Atributos:** `productoComboId`, `sku` propio del combo, `nombre/descripcion` del combo, lista de **ComponenteCombo** (cada componente con `productoReferenciadoId` y `cantidad`). Puede tener también un atributo calculado `pesoTotal` (suma de pesos de componentes) y posiblemente un `precioReferencia` (aunque normalmente los precios se gestionan en *ListaPreciosBC*).
- **Invariante:** La lista de componentes no puede estar vacía ni contener duplicados exactos. **No puede incluir combos** como componente (evita recursividad) ni incluir productos inactivos. La cantidad de cada componente debe ser al menos 1 (regla validada en ComponenteCombo).
- **Comportamiento:** Se crea vía **Caso de uso Crear Producto Combo** indicando los componentes. Internamente, valida para cada componente que el producto exista y esté activo. Puede calcular y almacenar su peso total sumando (`producto.peso * cantidad`) de cada componente (método `calcularPesoTotal()`). Antes de guardar, ejecuta `validarComponentesActivos()` para garantizar las invariantes. Al **Editar Ítem** (aplicable a combos también), si se cambian componentes, se revalida todo. Al **Deshabilitar** el combo, se marca inactivo; puede emitir *ProductoInhabilitado* para que facturación no lo use más. Si alguno de sus componentes es deshabilitado en el catálogo más adelante, podría dispararse una política para alertar o marcar el combo como inconsistente hasta corrección.

## 2.1.2 Entidades (dentro del contexto de los agregados)

Además de los agregados raíz descritos, existen entidades de apoyo sin vida independiente fuera de su agregado:

- **AtributoVariante:** Entidad (o VO complejo) que representa una característica específica de una variante. Tiene campos `nombreAtributo` (p.ej. "Color") y `valorAtributo` (p.ej. "Rojo"). En la implementación se trata como un **Objeto de Valor inmutable** (no cambia una vez creado, y se identifica por su par nombre-valor). Pertenece a *ProductoVariante* (no tiene repositorio propio). Su invariantes son triviales: nombre y valor no vacíos; y su igualdad se define por contenido (para detectar duplicados).
- **ComponenteCombo:** Entidad perteneciente a un *ProductoCombo* que detalla un componente del paquete. Posee `productoId` (referencia al producto componente, puede ser simple o variante) y `cantidad`. Podría tener un ID técnico interno, pero su identidad lógica es compuesta (producto + combo). **Reglas:** `cantidad > 0` (normalmente entero positivo) – método `validarCantidad()` lo asegura. No se permiten componentes repetidos (mismo producto dos veces), a menos que se modelen por separado sumando cantidades. Pertenece exclusivamente al combo; se crea, actualiza o elimina sólo en el contexto de gestionar el combo.

- **MultimediaProducto:** Entidad de adjunto multimedia asociada a un producto (sea simple, variante o combo). Tiene atributos como `multimediaId`, `tipoArchivo` (imagen, pdf, etc.), `urlArchivo` o contenido, y posiblemente `descripcion` o `tag`. Se asocia con el producto vía su ID. Las reglas de negocio controlan su contenido mediante *métodos de dominio*: **validarFormatoArchivo()** (por ejemplo, permitir solo imágenes JPEG/PNG hasta cierto tamaño, o PDFs para manuales) y la política de cuántos archivos se pueden asociar por producto. Aunque el almacenamiento real pueda delegarse a infraestructura, a nivel de dominio se considera la presencia de uno o varios *MultimediaProducto* ligados a un producto para fines de consistencia (p.ej., al clonar/duplicar un producto podría replicar referencias multimedia, o al eliminar un producto se eliminan sus adjuntos vía `eliminarAdjunto()`).

(Nota: En la implementación, *AtributoVariante* muy probablemente será una clase de valor simple dentro de la colección de la variante; *ComponenteCombo* y *MultimediaProducto* podrían mapearse a tablas secundarias vinculadas por foreign key al combo o producto, respectivamente, representando composición dentro del agregado.)

### 2.1.3 Objetos de Valor (VO) ↗

Ya enumerados en la sección 1.5.2, recalcamos los VO del dominio con sus **invariantes específicos** y uso:

- **UnidadMedidaVO:** Catálogo inmutable de unidades permitido. Invariante: debe ser uno de los códigos predefinidos (no se aceptan valores libres). Internamente puede contener código y quizás nombre/unidad larga. Es usado tanto para validación en *CatalogoArticulosBC* como para enviar la unidad correcta a SUNAT en la factura.
- **AfectacionIGVVO:** Enumeración de tipos de afectación de IGV. Invariante: valor debe pertenecer a {GRAVADO, EXONERADO, INAFECTO, etc.} listado oficialmente; además, podría tener lógica asociada (e.g., si es **GRAVADO**, indicar que el impuesto es 18% normalmente, aunque ese cálculo es fuera del catálogo).
- **CodigoSUNATVO:** Valor compuesto de 8 dígitos numéricos. Invariante: exactamente longitud 8 y debe existir en el catálogo oficial. Podría implementarse validando contra un conjunto de códigos cargados en memoria desde *ConfiguracionSistemaBC*.
- **CuentaContableVO:** Código alfanumérico de cuenta. Invariante: Debe concordar con alguna cuenta del plan contable configurado. Ejemplo: solo permitir dígitos o estructura específica si la empresa usa cierto formato. Este VO ayuda a evitar cuentas mal digitadas en cada producto.
- **CentroCostoVO:** Similar a CuentaContable, es un identificador predefinido. Invariante: Debe existir en el catálogo de centros de costo de la empresa.
- **PresupuestoVO:** Valor monetario (posible objeto Monetario con cantidad y moneda). Invariante: no negativo (un presupuesto asignado no puede ser negativo). Pueden existir reglas adicionales como: si un producto está bajo cierto plan, el presupuesto debe ser  $\leq X$ , pero eso sería específico.
- **PesoVO:** Valor decimal con unidad implícita en Kg. Invariante:  $\geq 0.0$ ; y si asignado a un producto intangible, debería ser 0. Este VO podría también usarse para combos (peso total calculado).

Todos estos VO, al ser inmutables y comparables por valor, garantizan consistencia cuando se pasan entre contextos. Por ejemplo, *UnidadMedida* "NIU" en Catálogo será exactamente el mismo código que entiende *ComprobantesElectonicosBC*. Se evita duplicidad de definiciones.

### 2.1.4 Servicios de Dominio ↗

Dentro de *CatalogoArticulosBC*, identificamos servicios de dominio que encapsulan lógica que **no pertenece a una sola entidad**. Ya mencionados en parte en 1.5.3, aquí se definen sus **métodos y contratos principales**:

- **ValidacionCatalogoService** (o **ServicioValidacionSKU**): Provee métodos para que otros contextos verifiquen productos. Por ejemplo: `validarSku(sku: String) -> DetalleProducto`.

- **Entrada:** SKU a validar.
- **Proceso:** Busca el producto por SKU en el **Repositorio** del catálogo. Si no existe o está inactivo, lanza error o responde “no válido”. Si es válido, compone un objeto **DetalleProducto** con los datos solicitados (p.ej., descripción, unidadMedida, afectacionIGV, peso, tipoProducto).
- **Salida:** DetalleProducto (VO o DTO) con campos necesarios para el consumidor.
- **Contrato de mensaje:** Podría exponerse vía una API REST ( GET /catalogo/validar/{sku} ) retornando JSON, o vía un evento de respuesta en caso de una consulta asíncrona. Este servicio actúa como **anticorrupción** para que otros módulos no accedan directamente a la base de datos del catálogo ni repliquen su lógica de validación.
- **ImportacionProductosService:** Maneja la lógica de importación masiva. Métodos:  
`importarDesdeExcel(archivoExcel) -> ResultadoImportacion .`
  - Lee el archivo Excel, por cada fila crea un comando de **Crear Producto** (simple, variante o combo según campos) o de **Actualizar Producto** si se detecta existente.
  - Valida cada entrada utilizando las mismas reglas del dominio (unicidad de SKU, datos requeridos).
  - Orquesta la persistencia a través de los **Repositorios** de los agregados correspondientes, usualmente en una transacción por bloque o registrando errores individuales sin abortar toda la importación.
  - Devuelve un resultado consolidado: cuántos productos creados, actualizados, errores (con detalles por fila).
  - *Contrato:* Este servicio podría ser llamado desde un Application Service al subir un archivo en la UI. Podría funcionar de forma síncrona (esperando a procesar todo) o asíncrona (encolar un trabajo de importación y luego publicar un evento de *ProductosImportados* al terminar). En cualquier caso, la lógica central reside aquí, reutilizable para distintas interfaces.
- **ExportacionProductosService:** Complementario al anterior, genera un Excel/CSV del catálogo. Método:  
`exportarProductos(filtroOpcional) -> Archivo .`
  - Recorre los repositorios de productos aplicando algún filtro (p.ej., activos solamente, o por categoría).
  - Crea un archivo (CSV, XLSX) con columnas representando los campos del producto.
  - Retorna el archivo o una URL de descarga.
  - *Nota:* Si bien es más una funcionalidad de aplicación, se puede encapsular como servicio para testear la lógica de formato y asegurarse que incluya todos los datos necesarios (por ejemplo, convertir códigos a nombres legibles si se desea).
- **GestionMultimediaService:** Encapsula operaciones sobre *MultimediaProducto*. Métodos:  
`agregarArchivo(productoId, archivo), eliminarArchivo(productoId, multimediaId) .`
  - **Agregar:** Valida tipo y tamaño del archivo ( `validarFormatoArchivo()` del VO), genera un identificador, asocia el archivo (posiblemente subiéndolo a un storage externo) y guarda un registro *MultimediaProducto* en el repositorio del producto.
  - **Eliminar:** Verifica que el recurso exista asociado al producto y procede a marcarlo como eliminado (y opcionalmente borrarlo del almacenamiento físico).
  - Este servicio podría interactuar con un **FileStorageAdapter** en la capa de infraestructura para la parte de guardar/bajar archivos, manteniendo el dominio independiente de la implementación de almacenamiento.
- **Integración con otros dominios (Servicios externos):**

- *API Service de Facturación:* Aunque el motor de facturación electrónica (ej. generación de XML, firmado digital, envío a SUNAT) reside en otro dominio, **CatalogoArticulosBC** garantiza que ese servicio reciba datos limpios. En algunos casos, podría haber un dominio service en catálogo que provea directamente un **formato** que requiere el API de facturación. Por ejemplo, `obtenerDetalleSunat(productId)` que devuelve el objeto listo para incrustar en el XML (con código SUNAT, unidad en el código esperado, descripción formateada). Este servicio se basaría en los VO ya validados para no repetir lógica.
- *Servicio de Identidad/Autenticación:* No es exactamente un servicio de dominio del catálogo, pero desde la perspectiva de **CatalogoArticulosBC** hay que respetar su contrato para saber **qué tenant/empresa y qué usuario** está ejecutando la operación. Esto se suele manejar en la capa de aplicación (ver sección 7 y 12), pero conceptualmente, podríamos considerar una **Policy** que llegue como evento: por ejemplo, si **IdentidadBC** emite `PermisosActualizados(usuario, roles)`, el catálogo podría reaccionar (aunque en práctica es poco común que necesite hacerlo).

En resumen, los servicios de dominio aseguran que las operaciones más complejas del catálogo (validaciones cruzadas, importación masiva, coordinación con externos) se realicen correctamente y de forma reutilizable, sin duplicar lógica en múltiples lugares.

### 2.1.5 Eventos de Dominio

**CatalogoArticulosBC** emite varios **Domain Events** para notificar a otros contextos de cambios importantes en los productos. Cada evento es inmutable y lleva un **payload** con la información necesaria para los suscriptores. A continuación se listan los eventos clave y sus detalles:

- **ProductoCreado** – Indica que se ha creado un nuevo producto en el catálogo (sea simple, variante o combo).
  - *Productor:* Caso de uso **Crear Producto** (Simple/Variante/Combo). El evento se publica una vez que la creación fue exitosa y comprometida en la base de datos.
  - *Payload:* Identificador del producto (`productId` o `productoServicioId`), el `sku` asignado, el `tipoProducto` (SIMPLE, VARIANTE, COMBO), `usuarioId` que realizó la acción (para auditoría) y `fechaCreacion`. Opcionalmente podría incluir un snapshot de algunos atributos (p.ej. nombre).
  - *Consumidores:* **ListaPreciosBC** (escucha para generar entradas en listas de precios o asignar precios base por defecto). **IndicadoresNegocioBC** (podría escuchar para empezar a considerar ese SKU en sus reportes, aunque no crítico). Posiblemente la **UI/Frontend** (para actualizar en tiempo real la lista visible de productos si se usa websockets).
- **ProductoModificado** – Se dispara cuando se edita un producto existente (cambio en sus atributos relevantes).
  - *Productor:* Caso de uso **Editar Ítem** (puede incluir editar simple, actualizar una variante, etc.).
  - *Payload:* Identificador del producto (`productId`), lista de `camposModificados` (nombres de los atributos que cambiaron, p.ej. ["descripcion","afectacionIGV"]], `usuarioId` que hizo el cambio, `fechaCambio`. Al incluir los campos cambiados, los consumidores pueden decidir si necesitan actuar (ej: si cambió afectación IGV, recalcular precios).
  - *Consumidores:* **ListaPreciosBC** (recalcula precios o reglas si algún cambio afecta costos, impuestos, unidad de medida, etc.). **IndicadoresNegocioBC** (posiblemente para actualizar descripciones de productos en reportes históricos, aunque usualmente se mantienen las originales). **UI** (para reflejar cambios en catálogos mostrados al usuario).
- **ProductoInhabilitado** – Notifica que un producto (o variante/combo) ha sido marcado como inactivo (deshabilitado) en el catálogo.

- *Productor:* Caso de uso **Deshabilitar Ítem**. Puede originarse al deshabilitar directamente un producto simple, o una variante individual, o un combo.
- *Payload:* `productId`, `usuarioId` que realizó la acción, `fechaCambio`, y un posible `motivo` o comentario de inactivación (por ejemplo “obsoleto”, “descontinuado”).
- *Consumidores:* **ComprobantesElectonicosBC** (para dejar de permitir seleccionar ese SKU en nuevas facturas – p. ej., el POS podría filtrar productos inactivos). **ListaPreciosBC** (para excluirlo de listas de precios vigentes o marcarlo no disponible en promociones). Potencialmente **InventarioBC** (si existiera, para detener movimientos de stock de ese ítem). **UI** (para actualizar vistas).
- (**Otros Eventos Internos:**) Si bien los tres anteriores son los principales eventos *outbound* del contexto, pueden existir eventos internos o de menor escala, por ejemplo:
  - *MultimediaAgregado* (cuando se adjunta un archivo a un producto, si se desea auditar o notificar).
  - *ProductosImportados* (resumen del resultado de una importación masiva, que quizá *IndicadoresNegocioBC* o un *AdminBC* escuche para log de actividades).
  - *ComboComponentesActualizados* (si se quisiera notificar cambios en la composición de un combo para que otros contextos reaccionen, aunque usualmente no necesario).

Cada evento tiene un **contrato de mensaje** definido. Por ejemplo, el **ProductoInhabilitado** podría publicarse en un broker (Kafka, RabbitMQ) con un contenido JSON como:

1 json

CopiarEditar

```
{
  "evento": "ProductoInhabilitado",
  "datos": {
    "productId": "PRD-1001",
    "estadoNuevo": "INACTIVO",
    "fechaCambio": "2025-07-04T17:48:00Z",
    "motivo": "Obsolescencia",
    "usuarioId": "USR-900"
  }
}
```

Los consumidores al recibirla ejecutan sus propias acciones (ej: *ComprobantesElectonicosBC* marcará ese ID como no vendible). La estructura de los eventos se versiona cuidadosamente para mantener compatibilidad en la integración.

(Cabe señalar que *CatalogoArticulosBC* también **consume** eventos/datos de otros contextos – ver sección 1.6 – aunque estrictamente hablando esos son eventos externos para este BC, no generados por él.)

## 2.2 Políticas de Negocio y Especificaciones

Además de las reglas puntuales (RN-CA-001...005) ya enumeradas, el dominio implementa **políticas y especificaciones** más generales que gobiernan su comportamiento. Estas encapsulan **reglas de decisión** de más alto nivel o criterios reutilizables en distintos casos de uso:

- **Políticas de Validación General:** Por ejemplo, una *Policy* podría dictar que “*No se podrá crear ni habilitar un producto sin que estén configurados los catálogos base (unidades, impuestos, cuentas)*”. Esto se traduciría en que el servicio de creación primero verifique vía *ConfiguracionSistemaBC* que existen al menos una unidad de medida y afectación IGV activas en el sistema, evitando crear productos con datos incompletos. Otra política podría ser “*Si un producto cambia su categoría de IGV de gravado a exonerado, notificar al administrador*”, implementada quizás como un evento de dominio adicional o una alerta interna. Son lineamientos que trascienden una regla individual y velan por la calidad global del catálogo.
- **Especificaciones Compuestas:** Se pueden modelar *Specifications* en el código para verificar combinaciones de reglas. Ejemplo: una Specification **EsProductoValidoParaFactura** que evalúa: SKU existe, producto activo, tiene precio asignado en ListaPrecios, stock disponible (si correspondiese) y cliente autorizado (si requiere permisos

especiales). Aunque algunas de esas verificaciones ocurren en otros BC, CatalogoArticulosBC contribuye con parte de esa spec (SKU existente/activo). Estas especificaciones permiten en el futuro componer reglas más complejas sin duplicar lógica, p.ej., para un chatbot que consulta si un producto está listo para vender.

- **Validaciones de Formato:** A nivel de dominio se vela por formatos correctos: SKU quizá deba seguir un patrón (ej: alfanumérico de 5-10 chars, sin espacios). Código SUNAT debe ser numérico de 8 dígitos. Podríamos ejemplificar un mensaje de error JSON al fallar una validación de formato, por ejemplo al intentar crear un producto con un SKU ya usado:

1

```
{ "error": "ValidacionCatalogoError", "detalles": [ {"campo": "sku", "mensaje": "El SKU 'ABC123' ya existe en el cat\u00e1logo (debe ser \u00f3nico)."}, {"codigoRegla": "RN-CA-001"} ] }
```

Este mensaje sigue una pol\u00e9tica de notificaci\u00f3n de errores donde se indica qu\u00e9 regla se viol\u00f3.

- **Tasas Fiscales y C\u00e1lculos:** Si bien el cat\u00e1logo no realiza c\u00e1lculos monetarios (eso recae en facturaci\u00f3n/precios), s\u00ed guarda la categor\u00eda de impuesto. Una pol\u00e9tica global podr\u00faa ser “*IGV por defecto 18% para categor\u00eda GRAVADO*” almacenada en Configuraci\u00f3n. CatalogoArticulosBC no aplica esa tasa pero asegura que la categor\u00eda est\u00e9 bien puesta para que ListaPreciosBC y ComprobantesBC apliquen la tasa correcta. Tambi\u00e9n, en combos, aunque los precios se manejan afuera, una pol\u00e9tica interna podr\u00faa ser que “*el precio de combo no debe exceder la suma de precios individuales*” (si el negocio lo requiere como regla para combos promocionales); esta verificaci\u00f3n podr\u00faa hacerse en un servicio de dominio consultando ListaPreciosBC durante la creaci\u00f3n del combo.
- **Pol\u00e9tica Multi-Tenant:** A nivel de arquitectura SaaS, existe la pol\u00e9tica de **segregaci\u00f3n de datos por empresa**. Esto implica que todas las consultas y comandos en el cat\u00e1logo llevan un contexto de tenant (empresaId). En el dominio, esto se refleja en que los Repositorios filtran por tenant autom\u00e1ticamente, y en que no se permiten operaciones que crucen tenants (por ejemplo, no se puede asociar un producto de una empresa a un combo de otra). Esta pol\u00e9tica es implementada t\u00f3picamente en la capa de aplicaci\u00f3n/repositorio, pero forma parte de las reglas de negocio de la soluci\u00f3n.

CatalogoArticulosBC aplica un conjunto robusto de pol\u00e9ticas y especificaciones para mantener **consistente y alineado** el comportamiento del sistema con las reglas del negocio. Estas reglas aseguran que, sin importar c\u00f3mo evolucione el sistema (nuevos canales, integraciones o casos de uso), los principios fundamentales – unicidad de datos, validez tributaria, integridad relacional (padres-hijos), segregaci\u00f3n por empresa, etc. – se conserven.

## Casos de Uso\_ca

A continuación se describen detalladamente los principales casos de uso del BC, siguiendo el formato definido. Incluyen flujos de interacción paso a paso, excepciones y diagramas de secuencia.

### 3.1 Crear Producto Simple

**Descripción:** Registra un nuevo **ProductoSimple** con todos sus atributos obligatorios. El actor (`Usuario del sistema autenticado`) proporciona los datos básicos del producto.

#### Flujo principal:

1. El usuario ingresa los datos del producto: `sku, nombre, unidadMedida, afectacionIGV, codigoSUNAT, cuentaContable, centroCosto, presupuesto, peso, estado`.
2. El sistema valida la **unicidad del SKU** (RN-CA-001) y verifica que cada campo respete los formatos de los VOs.
3. Se crea un nuevo objeto `ProductoSimple` con dichos atributos y se asigna la `fechaCreacion` actual.
4. El sistema publica el evento **ProductoCreado** (con `tipoProducto=SIMPLE`).
5. Se retorna al usuario el `productoServicioId` generado y confirmación de éxito.

#### Flujos Alternativos:

- Si el SKU ya existe, se lanza una excepción `SKUDuplicadoException`.
- Si algún VO es inválido (p.ej. formato incorrecto), se lanza una excepción de validación correspondiente.

#### Diagrama de secuencia (Crear Producto Simple):



1 `mermaid`

### 3.2 Crear Producto Variante

**Descripción:** Crea un **ProductoVariante** asociado a un *ProductoSimple* existente. El usuario ingresa los atributos específicos de la variante (p.ej. color, talla).

**Precondiciones:** El `productoPadreId` debe existir y estar en estado ACTIVO; la combinación de atributos de variante debe ser nueva (no duplicada).

#### Flujo principal:

1. El usuario proporciona: `skuVariante, productoPadreId, listaAtributosVariante` (colección de {nombreAtributo, valorAtributo}).
2. El sistema verifica la existencia y estado `ACTIVO` del producto padre, y comprueba que no exista ya una variante con esa misma lista de atributos.
3. El sistema hereda los atributos comunes del padre (p.ej. descripción, unidad, afectaciónIGV) y asigna los nuevos atributos diferenciadores.
4. Se crea el objeto `ProductoVariante` con `fechaCreacion` actual y se publica el evento **ProductoCreado** (con `tipoProducto=VARIANTE`).
5. Se confirma la operación al usuario.

#### Flujos Alternativos:

- Si el padre está inactivo, se lanza *InvalidStateException*.
- Si la variante ya existe (mismos atributos), se lanza *VarianteDuplicadaException*.

#### Diagrama de secuencia (Crear Producto Variante):



```
1 mermaid
```

### 3.3 Crear Producto Combo

**Descripción:** Define un **ProductoCombo** que agrupa varios ítems (componentes). El usuario especifica el SKU del combo, sus componentes con cantidades, el precio total, unidad de medida, afectación IGV y estado.

**Precondiciones:** Todos los productos/variantes incluidos deben existir y estar activos; las cantidades deben ser  $\geq 1$ .

#### Flujo principal:

1. El usuario envía: *skuCombo*, *nombreCombo*, *lista de componentes (productoServicioId, cantidad)*, *precioCombo*, *unidadMedida*, *afectacionIGV*, *estado*.
2. El sistema valida la unicidad del SKU y confirma que cada componente existe y está **ACTIVO**.
3. Calcula el *pesoTotal* del combo sumando los pesos de los componentes.
4. Crea **ProductoCombo** con su lista de **ComponenteCombo**, asigna *fechaCreacion* y publica el evento **ProductoCreado** (con *tipoProducto=COMBO*).
5. Se confirma la creación al usuario.

#### Flujos Alternativos:

- Si algún componente está inactivo, se lanza *ProductoPadreInvalidoException*.
- Si alguna cantidad  $< 1$ , se lanza *CantidadInvalidaException*.

#### Diagrama de secuencia (Crear Producto Combo):



```
1 mermaid
```

### 3.4 Editar Ítem

**Descripción:** Actualiza los atributos de un ítem existente (simple, variante o combo). El usuario especifica el *productoServicioId* y los campos a modificar.

**Precondiciones:** El *productoServicioId* debe existir y estar en estado **ACTIVO**.

#### Flujo principal:

1. El usuario envía *productoServicioId* y lista de *camposModificados* con sus nuevos valores.
2. El sistema valida que el producto exista y esté **ACTIVO**.
3. Se aplican los cambios al objeto (actualizando los atributos correspondientes), se registra el cambio en **HistorialProducto** para auditoría y se actualiza la *fechaModificacion*.
4. Se publica el evento **ProductoModificado** con los detalles del cambio.
5. Confirma el éxito de la operación al usuario.

#### Flujos Alternativos:

- Si el ítem está inactivo, se lanza *InvalidStateException*.
- Si algún atributo es inválido, se lanzan excepciones de validación de VO.

#### Diagrama de secuencia (Editar Ítem):



### 3.5 Deshabilitar Ítem

**Descripción:** Marca un ítem existente como **INACTIVO**, impidiendo su uso futuro. El usuario proporciona el `productoServicioId` y el motivo.

**Precondiciones:** El `productoServicioId` debe existir. Para productos simples y variantes, además no deben estar incluidos en combos activos.

#### Flujo principal:

1. El usuario envía `productoServicioId` y `motivoDeshabilitacion`.
2. El sistema valida que no haya dependencias activas (p.ej. el producto no forma parte de un combo activo).
3. Cambia el estado del producto a **INACTIVO** y registra la razón.
4. Publica el evento **ProductoInhabilitado** con `productoServicioId`, `usuarioId`, `fechaInhabilitacion` y `motivo`.
5. Informa al usuario que se ha deshabilitado exitosamente.

#### Flujos Alternativos:

- Si existen dependencias activas, se lanza *DependenciaActivaException*.

#### Diagrama de secuencia (Deshabilitar Ítem):



1 mermaid

### 3.6 Gestionar Multimedia

**Descripción:** Agregar o eliminar recursos multimedia asociados a un ítem. El sistema mantiene una colección de `MultimediaProducto` por producto.

**Precondiciones:** El `productoServicioId` o `multimediaId` debe existir.

#### Flujo principal (Aregar):

1. El usuario envía `productoServicioId`, `tipoAdjunto`, `nombreArchivo`, `ruta`, `comentario`.
2. El sistema valida el tipo de archivo (imagen, PDF, etc.) y el peso máximo permitido.
3. Crea un nuevo objeto `MultimediaProducto` vinculado al producto y publica **ProductoModificado** (notificando la adición).
4. Confirma la operación al usuario.

#### Flujo principal (Eliminar):

1. El usuario envía `multimediaId`.
2. El sistema valida la existencia del recurso multimedia y elimina el registro asociado.
3. Publica **ProductoModificado** (para indicar que el catálogo cambió).
4. Retorna confirmación.

#### Flujos Alternativos:

- Si el archivo es inválido (tipo o peso), se lanza *MultimediaInvalidaException*.

#### Diagrama de secuencia (Gestionar Multimedia – Agregar):



#### Diagrama de secuencia (Gestionar Multimedia – Eliminar):



### 3.7 Importar Productos desde Excel

**Descripción:** Procesa un archivo (CSV/Excel) para crear o actualizar ítems en lote. Se lanza una operación asíncrona tras subir el archivo.

**Precondiciones:** El archivo debe contener todas las columnas obligatorias (sku, tipoProducto, etc.).

#### Flujo principal:

1. El usuario sube el archivo de productos al sistema.
2. El sistema valida la estructura general (columnas necesarias) y procesa el archivo fila por fila:
  - Si el sku de la fila ya existe, invoca internamente el Caso de Uso **Editar Ítem** (3.4) con los campos actualizados.
  - Si el sku no existe, invoca el Caso de Uso **Crear Producto Simple/Variante/Combo** correspondiente (1–3) según el campo **tipoProducto** de la fila.
3. El sistema genera un resumen de éxitos y errores al finalizar.
4. Para cada fila procesada con éxito, se publica el evento **ProductoCreado** o **ProductoModificado** según corresponda.

#### Flujos Alternativos:

- Filas con error (datos inválidos, excepciones) son omitidas y registradas en el informe final.

#### Diagrama de secuencia (Importar Productos):



### 3.8 Exportar Productos a Excel

**Descripción:** Genera un archivo (CSV/Excel) que contiene todos los atributos de los ítems del catálogo, para descarga externa. El usuario puede aplicar filtros opcionales.

**Precondiciones:** Los filtros proporcionados (categoría, estado, etc.) deben ser válidos.

#### Flujo principal:

1. El usuario selecciona filtros (p.ej. categoría, familia, estado) y solicita la exportación.
2. El sistema recupera los ítems que cumplen los criterios y construye el archivo Excel.
3. Se genera un enlace o descarga inmediata del archivo al usuario.

No se generan eventos ni se modifican entidades (solo lectura).

#### Diagrama de secuencia (Exportar Productos):



### 3.9 Consultar Producto por SKU

**Descripción:** Obtiene la ficha detallada de un producto dado su SKU. Puede ser invocado por el UI o por el subsistema de facturación.

**Precondiciones:** El sku debe corresponder a un producto existente.

#### Flujo principal:

1. El actor solicita el producto pasando el *sku* como parámetro.
2. El sistema recupera la entidad raíz ( *ProductoSimple* , *ProductoVariante* o *ProductoCombo* ) correspondiente y carga sus VOs relacionados (p.ej. *AtributoVariante*, *ComponenteCombo*).
3. Se devuelve un DTO al llamador con toda la información del producto (campos básicos y colecciones asociadas).

**Flujos Alternativos:**

- Si no se encuentra el SKU, se lanza *EntityNotFoundException*.

**Diagrama de secuencia (Consultar Producto por SKU):**



### 3.10 Listar Productos

**Descripción:** Obtiene una lista paginada de ítems del catálogo, aplicando filtros (categoría, familia, estado, etc.). El usuario define parámetros de paginación y filtros.

**Precondiciones:** Parámetros de página y filtros válidos (p. ej. página  $\geq 1$ , tamaño dentro de límites).

**Flujo principal:**

1. El usuario envía los filtros y parámetros de paginación (página, tamaño).
2. El sistema aplica los filtros en la base de datos y recupera la página solicitada de ítems.
3. Se devuelve un conjunto de DTOs resumidos con información básica de cada producto (SKU, nombre, categoría, estado, etc.).

No se generan eventos ni se modifican entidades (consulta de solo lectura).

**Diagrama de secuencia (Listar Productos):**



### 3.11 Validar SKU

**Descripción:** Provee un servicio de consulta para validar un SKU y obtener sus atributos esenciales. Este caso de uso es típicamente invocado por subsistemas externos (p.ej. facturación) para verificar datos de producto.

**Precondiciones:** El *sku* debe existir en el catálogo.

**Flujo principal:**

1. El subsistema invoca el servicio con el *sku* solicitado.
2. El sistema busca el producto asociado y retorna sus datos clave: *productoServicioId*, *nombre*, *unidadMedida*, *afectacionIGV*, *codigoSUNAT*, *peso* (pudiendo incluir otros campos si es necesario).
3. Si el SKU no existe, retorna error *EntityNotFoundException*.

**Diagrama de secuencia (Validar SKU):**



## Eventos de Dominio\_ca

A continuación se listan y documentan los eventos de dominio generados por este BC:

- **ProductoCreado:**

- *Payload:*

```
1 {
2   "productoServicioId": "UUID",
3   "sku": "String",
4   "tipoProducto": "SIMPLE|VARIANTE|COMBO",
5   "usuarioId": "UUID",
6   "fechaCreacion": "Timestamp"
7 }
```

- *Productor:* Casos de uso *Crear Producto Simple*, *Crear Producto Variante*, *Crear Producto Combo*.

- *Consumidor:* ListaPreciosBC (actualiza listas de precios base), IndicadoresNegocioBC (inicia métricas de ventas por SKU), interfaz de usuario (mostrar nuevo producto).

- **ProductoModificado:**

- *Payload:*

```
1 {
2   "productoServicioId": "UUID",
3   "camposModificados": [
4     { "campo": "String", "valorAnterior": "String", "valorNuevo": "String" }
5     ...
6   ],
7   "usuarioId": "UUID",
8   "fechaModificacion": "Timestamp"
9 }
```

- *Productor:* Casos de uso *Editar Ítem* y *Gestionar Multimedia (Agregar/Eliminar)*.

- *Consumidor:* ListaPreciosBC (recalcula precios si cambian tributos fiscales), IndicadoresNegocioBC (refresca métricas si cambia categoría/atributos), UI (actualizar vista de detalles).

- **ProductoInhabilitado:**

- *Payload:*

```
1 {
2   "productoServicioId": "UUID",
3   "usuarioId": "UUID",
4   "fechaInhabilitacion": "Timestamp",
5   "motivoDeshabilitacion": "String"
6 }
```

- *Productor:* Caso de uso *Deshabilitar Ítem*.
- *Consumidor:* ComprobantesElectonicosBC (rechaza futuros comprobantes con ese SKU), ListaPreciosBC (remueve el ítem de listas activas), UI (marcar ítem como inactivo en catálogos).

Cada evento se publica en forma de mensaje JSON estructurado como se muestra, utilizando un *contract* predefinido. Por ejemplo, el contrato JSON de `ProductoCreado` incluiría los campos arriba descritos. Los consumidores deben poder deserializar estos mensajes para reaccionar en consecuencia, manteniendo desacoplados los dominios.

# Restricciones y Reglas de Negocio\_ca

## 1. Contexto

CatalogoArticulosBC aplica controles para garantizar la integridad de datos del catálogo y el cumplimiento de normativas tributarias y operativas.

## 2. Definiciones

- Reglas de Negocio:** Condiciones lógicas o normativas que definen cómo debe comportarse el sistema para cumplir requerimientos de negocio y regulatorios. Se validan durante la ejecución de casos de uso.
- Restricciones:** Limitaciones operativas o técnicas que impiden iniciar o completar una acción si no se cumplen criterios previos. Se aplican antes o al inicio de la operación.

## 3. Diferencia entre Reglas y Restricciones

Concepto	Rol en el sistema	Se aplica cuando...	Consecuencia si no se cumple
Regla de Negocio	Define comportamiento en flujo de Caso de Uso	Durante la ejecución del caso de uso	Rechaza la acción o registra auditoría
Restricción	Define cuándo NO debe permitirse una operación	Antes o al inicio del caso de uso	Bloquea la ejecución con error especializado

---

## 4. Condiciones de negocio por Caso de Uso

### Caso de Uso 1 – Crear Producto Simple

- Entidad base:** ProductoSimple
- Rango de códigos:** RN-CA-001 a RN-CA-004

#### Restricciones:

- SKU debe ser único.
- Usuario con permiso CatalogoArticulos.Crear .

#### Reglas de Negocio Aplicadas:

Código	Regla
RN-CA-001	Verificar unicidad de SKU.

RN-CA-002	Validar CódigoSUNAT contra catálogo oficial.
RN-CA-003	UnidadMedida válida según ConfiguracionSistemaBC.
RN-CA-004	Peso > 0 para productos físicos.

#### Caso de Uso 2 – Crear Producto Variante

- **Entidad base:** ProductoVariante
- **Rango de códigos:** RN-CA-001, RN-CA-005

#### Restricciones:

- Producto padre debe estar ACTIVO .
- Combinación de atributos de variante no puede existir.

#### Reglas de Negocio Aplicadas:

Código	Regla
RN-CA-001	Validar existencia y estado de producto padre.
RN-CA-005	Evitar duplicación de combinación de atributos de variante.

#### Caso de Uso 3 – Crear Producto Combo

- **Entidad base:** ProductoCombo
- **Rango de códigos:** RN-CA-005, RN-CA-006

#### Restricciones:

- Todos los componentes deben estar ACTIVO .
- No incluir combos anidados.

#### Reglas de Negocio Aplicadas:

Código	Regla
RN-CA-005	Verificar que componentes estén activos y no sean combos.
RN-CA-006	Cantidad de cada componente $\geq 1$ .

#### Caso de Uso 4 – Editar Ítem

- **Entidad base:** ProductoSimple/Variante/Combo
- **Rango de códigos:** RN-CA-001 a RN-CA-004, RN-CA-005

##### Restricciones:

- Ítem debe estar ACTIVO .
- SKU no puede modificarse a duplicado.

##### Reglas de Negocio Aplicadas:

Código	Regla
RN-CA-001	Mantener unicidad de SKU al editar.
RN-CA-003	Verificar UnidadMedida después de cambio.
RN-CA-005	Confirmar validez de variantes y componentes al editar combo.

---

#### Caso de Uso 5 – Deshabilitar Ítem

- **Entidad base:** ProductoSimple/Variante/Combo
- **Rango de códigos:** RN-CA-006

##### Restricciones:

- Ítem si puede estar referenciado en comprobantes anteriores.

##### Reglas de Negocio Aplicadas:

Código	Regla
RN-CA-006	Verificar ausencia de referencias en combos antes de inhabilitar.

---

#### 5. Cuadro de Control de Códigos de Reglas de Negocio

Rango de Códigos	Entidad / Tema	Descripción del Grupo	Total Definidas	Próx. Código	Observaciones
RN-CA-001–RN-CA-006	CatalogoArticulosBC	SKU, tributos, unidades y estado de ítems	6	RN-CA-007	Reservado para reglas de importación masiva

Este BC aplica las siguientes invariantes y reglas críticas:

- **SKU único:** Ningún otro ítem en el catálogo puede tener el mismo SKU. Esto se verifica al crear o editar productos.
- **Variantes:** Sólo se pueden crear variantes de un producto si el producto padre está en estado ACTIVO . Además, cada combinación de atributos de variante debe ser única.
- **Combos:** Un combo no puede incluir ítems INACTIVOS ni otros combos (sin anidamiento). Todos los componentes deben existir y estar activos.
- **Código SUNAT válido:** El valor `codigoSUNAT` asignado debe existir en el catálogo oficial de la SUNAT (Regla RN-CA-001).
- **Unidad de medida válida:** El `unidadMedida` debe pertenecer al catálogo vigente de unidades.
- **Peso de productos:** Para productos físicos, `peso` debe ser un número decimal mayor que cero (Regla RN-CA-003).
- **Afectación IGV:** El valor `afectacionIGV` debe ser una de las opciones permitidas (GRAVA, EXONERADO, INAFECTO, etc.).
- **Soft delete:** Se usa borrado lógico: un producto se inactiva en lugar de eliminarse completamente. No se permite eliminar productos con historial de transacciones (RN-CA-005).
- **Multimedia:** Archivos adjuntos deben cumplir límites de tamaño y tipos MIME predefinidos (p.ej. imágenes JPEG/PNG hasta 5MB).

Si alguna de estas políticas se viola durante un caso de uso, el sistema interrumpe la operación lanzando excepciones especializadas y registrando auditoría. Estas reglas refuerzan la integridad de los datos del catálogo y aseguran el cumplimiento de normativas fiscales y operativas dentro del BC.

## Lenguaje Ubicuo\_ca

**Entidades del Dominio:** En el catálogo de artículos la entidad principal es **Artículo**, con identidad propia y atributos clave como *código, nombre, unidad de medida, categoría*, y posiblemente *precio* (aunque el precio puede manejarlo el BC de ListaPrecios). También existirían entidades secundarias según necesidad, por ejemplo **CategoríaArtículo** (u objeto de valor si es simple) o **UnidadMedida**. Cada artículo vive en un contexto multiempresa/multitenant, por lo que incluye identificadores de tenant/empresa.

**Objetos de Valor:** Los VOs describen propiedades del artículo que no tienen identidad propia. Ejemplos son **CodigoArticulo**, **NombreArticulo**, **UnidadMedida**, **CategoríaArtículo**, o incluso **Precio** (si el precio base se modela aquí). Estos objetos son inmutables y definidos por sus atributos. Por ejemplo, un **CodigoArticulo** podría ser una cadena única, y **UnidadMedida** un texto predefinido.

**Comandos:** Operaciones que realizan cambios en el catálogo. Ejemplos de comandos (en infinitivo) incluyen:

- **CrearArticuloCommand:** datos necesarios para dar de alta un nuevo artículo (tenantId, empresaId, código, nombre, unidad, categoría, etc.).
- **ActualizarArticuloCommand:** modifica campos de un artículo existente.
- **EliminarArticuloCommand:** borra un artículo del catálogo.
- **AsignarCategoriaArticuloCommand:** asigna o cambia la categoría de un artículo.
- **ActualizarStockArticuloCommand** (si se gestiona stock).

**Eventos de Dominio:** Hechos relevantes publicados tras procesar un comando exitosamente. Ejemplos:

- **ArticuloCreadoEvent, ArticuloActualizadoEvent, ArticuloEliminadoEvent, CategoriaArticuloAsignadaEvent**, etc. Estos eventos son emitidos por los agregados tras cambiar su estado. El patrón de Evento de Dominio dicta que los agregados publican estos eventos para notificar a otros servicios.

A continuación, una tabla con ejemplos de comandos y eventos comunes:

Comando	Evento Emitido	Descripción
<b>CrearArticuloCommand</b>	<b>ArticuloCreadoEvent</b>	Crea un artículo nuevo en el catálogo.
<b>ActualizarArticuloCommand</b>	<b>ArticuloActualizadoEvent</b>	Actualiza datos de un artículo existente.
<b>EliminarArticuloCommand</b>	<b>ArticuloEliminadoEvent</b>	Elimina un artículo del catálogo.
<b>AsignarCategoriaArticuloCommand</b>	<b>CategoriaArticuloAsignadaEvent</b>	Cambia la categoría asignada al artículo.

Por ejemplo, un comando JSON para crear artículo podría ser:

```
1 {
2   "tenantId": "T123",
3   "empresaId": "E456",
```

```
4   "codigo": "ART-001",
5   "nombre": "Artículo de Ejemplo",
6   "unidad": "Unidad",
7   "categoria": "CAT01"
8 }
```

Y su evento resultante `ArticuloCreadoEvent` podría verse así:

```
1 {
2   "articuloId": "A789",
3   "tenantId": "T123",
4   "empresaId": "E456",
5   "codigo": "ART-001",
6   "nombre": "Artículo de Ejemplo",
7   "unidad": "Unidad",
8   "categoria": "CAT01",
9   "timestamp": "2025-07-05T22:00:00Z"
10 }
```

Estos ejemplos usan el lenguaje ubicuo del dominio (**Artículo**, **Código**, **Categoría**, etc.). El uso de comandos y eventos respeta prácticas DDD: un comando encapsula la intención de cambio, y cada agregado publica un evento al actualizar su estado. Esto permite que otros bounded contexts (como `ListaPrecios` o `ControlCaja`) reaccionen a los cambios de catálogo suscribiéndose a estos eventos.

## 6.2 Conversaciones Ubiqutario ☀

- **Usuario (dueño de microempresa):** “Necesito agregar un artículo nuevo con su código y categoría.”
- **Desarrollador:** “Claro. En nuestro dominio tenemos la entidad `Artículo`, que tiene atributos como código (tipo `CodigoArticulo`), nombre ( `NombreArticulo` ), unidad ( `UnidadMedida` ), y referencia a la categoría ( `CategoríaArticulo` ). Para crearlo emitiremos un `CrearArticuloCommand` .”
- **Usuario:** “¿Y cómo sabe el sistema las reglas de negocio?”
- **Desarrollador:** “El agregado `Artículo` asegurará la unicidad del código dentro de la empresa y validará sus invariantes. Si todo está bien, publicamos un evento `ArticuloCreadoEvent` para notificar a otros servicios (como el catálogo de precios).”
- **Usuario:** “Perfecto, entonces esos términos (**Artículo**, **Código**, `CrearArticuloCommand`, `ArticuloCreadoEvent`) son el lenguaje común que usamos.”
- **Desarrollador:** “Exacto, usamos estos nombres en todo el diseño, código y conversaciones para evitar confusiones.”

Este diálogo exemplifica cómo se comparte la terminología (lenguaje ubicuo) entre el negocio y el equipo técnico, usando los conceptos DDD definidos.

## Servicios de Aplicación\_ca

Los *Application Services* definen los casos de uso y orquestan la lógica entre la interfaz (APIs, UI) y el dominio. Son “responsables del flujo principal de la aplicación”: coordinan entidades, objetos de valor, servicios de dominio e infraestructura para ejecutar cada caso de uso. Por ejemplo, un servicio de aplicación **ArticuloAppService** contendría métodos para `crearArticulo`, `actualizarArticulo`, `eliminarArticulo`, `obtenerArticulo`, etc. Cada método recibe un comando (o parámetros simples), llama a las entidades/agg y repositorios correspondientes, y devuelve un DTO o confirma la acción.

Por ejemplo, la interfaz podría definirse así (en pseudocódigo Java/Spring):

```
1 public interface ArticuloAppService {  
2     ArticuloDto crearArticulo(CrearArticuloCommand cmd);  
3     ArticuloDto actualizarArticulo(ActualizarArticuloCommand cmd);  
4     void eliminarArticulo(EliminarArticuloCommand cmd);  
5     ArticuloDto obtenerArticuloPorId(String articuloId);  
6 }
```

- **Ejemplo de método:** `crearArticulo(CrearArticuloCommand cmd)` toma el comando con datos del nuevo artículo, invoca la creación de la entidad (posiblemente a través de un *Factory* para validar el estado inicial) y usa el *Repository* para persistirlo. Al final publica el evento de dominio correspondiente. Como respuesta, podría devolver un `ArticuloDto` con la información creada o el ID generado.

Las implementaciones concretas de estos servicios coordinan los llamados a repositorios e infraestructuras (bases de datos, colas de eventos) sin contener lógica de negocio compleja: esa queda dentro de las entidades y dominios. Su responsabilidad es orquestar el flujo end-to-end del caso de uso.

Por ejemplo, métodos de la API REST podrían traducir peticiones JSON a comandos, pasarlo a `ArticuloAppService.crearArticulo()`, y luego retornar respuesta JSON. A continuación un ejemplo de contrato de servicio (tipo RESTful) con parámetros y respuesta (simplificado):

- **POST /api/catalogo/articulos**
  - *Request body:* datos del comando `CrearArticuloCommand`.
  - *Response:* `{"articuloId": "A789", "codigo": "ART-001", "nombre": "Artículo de Ejemplo", ...}` (DTO del artículo creado).
- **GET /api/catalogo/articulos/{id}**
  - *Response:* JSON del `ArticuloDto` con todos sus atributos.

Según DDD y CQRS, los servicios de aplicación sólo manejan escritura/actualización aquí; las consultas podrían ir a un QueryService separado o directamente al repositorio de lectura.

## Infraestructura y Adaptadores\_ca

La capa de infraestructura implementa los detalles técnicos (persistencia, comunicación, etc.) sin contaminar el dominio.

En este BC se contemplan:

- **Repositorios:** Implementan interfaces de acceso a datos definidas en el dominio. Por ejemplo, `ArticuloRepository` puede usar JPA/Hibernate (SQL) o MongoDB para almacenar artículos. Cada repositorio es responsable de cargar y persistir agregados completos (por ejemplo, un `Articulo` junto con sus atributos). En una arquitectura multitenant, se usarían esquemas separados o filtros por `tenantId` para aislar los datos de cada empresa.
- **Gateways/Adapters:** Si el catálogo requiere interactuar con servicios externos (p.ej. obtener datos de categorías desde Configuración de Sistema, o validar datos con Identity), se crearían adaptadores de salida (por ejemplo, `CategoriaGateway`, `AutenticacionGateway`) que llamen a esas APIs externas. Estos adaptadores cumplen el patrón *Gateway*, traduciendo entre nuestro modelo y la API externa.
- **Event Bus / Mensajería:** Para publicar eventos de dominio hacia otros BC (como `ListaPrecios` o `ControlCaja`), se usaría un bus de eventos (Kafka, RabbitMQ, etc.). Cada vez que el agregado `Articulo` emite un evento (por ejemplo, `ArticuloCreadoEvent`), un manejador de dominio publica ese evento en el bus para que otros servicios lo consuman. También podrían existir *suscriptores* (event handlers) para reaccionar a eventos de otros BC (p.ej. actualizar stock ante ventas).
- **APIs Externas:** El BC expone APIs REST bien documentadas (Open Host Service) que el Orquestador de Apps y otros BC consumen directamente. Por ejemplo, el endpoint POST `/api/catalogo/articulos` es un Open Host Service diseñado para uso de downstreams

### Patrones y Roles:

- **Factory:** Clases como `ArticuloFactory` encapsulan la lógica de creación de agregados complejo. Por ejemplo, el factory puede validar el comando y construir un `Articulo` en estado válido (asignar código, generar ID, inicializar atributos). Esto mantiene el constructor del dominio limpio.
- **Mapper:** En infraestructura, un *mapper* convierte entre objetos de dominio y DTOs o entidades JPA. Por ejemplo, `ArticuloEntity` (JPA) vs. `Articulo` (dominio), o `ArticuloDto` (respuesta). Puede usarse bibliotecas como MapStruct. El mapper no añade lógica de dominio, sólo mapea propiedades.
- **Handler:** En un esquema CQRS/Eventos, cada comando o evento puede tener un *handler*. Por ejemplo, un `CrearArticuloHandler` recibe el comando, llama al servicio de aplicación, y otro `ArticuloCreadoEventHandler` podría actualizar lecturas o notificar otros sistemas. Estos handlers cumplen el patrón *CommandHandler/EventHandler* coordinando los mensajes entrantes.
- **Transactional Outbox:** Para garantizar que los eventos se publiquen sólo si la transacción de base de datos se confirma, puede usarse el patrón *Transactional Outbox*: primero guardar el evento en una tabla y luego otro proceso lo envía al bus, evitando inconsistencias.

En síntesis, la infraestructura implementa repositorios para persistencia, adaptadores para APIs/colas, y patrones básicos (Factory, Repository, Handlers, Mappers) para separar responsabilidades. El dominio de artículos permanece limpio de detalles técnicos.

## Mapa de Contextos\_ca

El *Context Map* muestra cómo se relaciona el BC de Catálogo de Artículos con otros bounded contexts y sistemas. En nuestro diagrama (Mermaid) se esquematizan las conexiones principales:



- **CatalogoArticulosBC – ListaPreciosBC:** La relación es *Customer-Supplier*. Catalogo es proveedor (upstream) de datos de artículos (código, descripción, etc.); ListaPrecios es consumidor (downstream) que define precios para cada artículo. El catálogo publica eventos de artículo (p.ej. `ArticuloCreadoEvent`) y el BC ListaPrecios los consume para actualizar sus listas. Pueden usar un *lenguaje publicado* común (JSON de evento).
- **CatalogoArticulosBC – ControlCajaBC (Caja registradora/TPV):** Relaciones similares de *Customer-Supplier*. ControlCaja necesita datos del artículo al facturar ventas. El catálogo expone un API bien documentado (Open Host Service) y/o envía eventos de actualización de artículos; Caja lee información actual de cada artículo (descripción, unidad, stock) al generar tickets. Cambios en catálogo influyen en Caja, pero no viceversa (upstream/downstream).
- **CatalogoArticulosBC – GestionClientesBC:** Son *contextos separados* (Separate Ways). No hay interacción directa entre clientes y catálogo de artículos; gestion clientes se encarga de personas/jurídicos y no afecta la definición de artículos.
- **CatalogoArticulosBC – ConfiguracionSistemaBC:** No hay relación directa fuerte (Separate Ways). Sólo podrían compartir datos genéricos (p.ej. lista de categorías globales) si se decide ubicar en Configuración. Si existiera dependencia, sería vía consulta a una API externa.
- **CatalogoArticulosBC – ComprobantesElectonicosBC:** Patrón *Published Language*. El BC de ComprobantesElectonicos (facturación) requiere datos de artículo para generar líneas de factura (descripción, unidad, impuestos). Puede suscribirse a los eventos del catálogo (artículo creado/actualizado) usando mensajes JSON definidos como lenguaje común. Así, ambas BCs usan el mismo formato de mensaje para intercambio.
- **CatalogoArticulosBC – Mi Contasis (Identidad/Auth):** *Open Host Service*. El sistema de identidad es un proveedor externo de autenticación/autorización. El catálogo autentica peticiones verificando tokens JWT emitidos por Mi Contasis. En el contexto, Identity es upstream (provee identidad) y actúa vía API/Token. Es un OHS: se exponen servicios de identidad que consumo downstream
- **CatalogoArticulosBC – Orquestador de Apps:** *Open Host Service / Downstream*. El orquestador central coordina flujos multi-servicio (p.ej. crear empresa, configurar todo). Llama a las APIs de Catálogo (y otros BC) para ejecutar tareas compuestas. No comparte modelo con el catálogo, sino que consume sus endpoints; por tanto funciona como cliente downstream (similar a un *Published Language* si fueran eventos, pero aquí más flujo de comandos).
- **Otros vínculos internos (ListaPrecios–ControlCaja, ControlCaja–Comprobantes, etc.):** Se indican en el diagrama. Por ejemplo, ListaPrecios provee precios a Caja (Customer-Supplier) y Caja informa ventas a Comprobantes (Customer-Supplier). Estos detallan la topología global del sistema.

En resumen, los patrones usados son **Customer-Supplier** (cadena de suministro de datos), **Open Host Service** (APIs bien documentadas), y **Published Language** para intercambio mediante eventos/JSON comunes. No se usa **Shared Kernel** ni

*Conformist* en este caso, ya que no hay código compartido ni presión para adoptar modelos de otro BC. Cada contexto es relativamente independiente y se comunica por servicios/colas.

## Política de Consistencia y Transaccionalidad\_ca

Para mantener la integridad de los datos en este entorno distribuido, adoptamos **consistencia eventual** mediante el patrón Saga. No se usan transacciones ACID globales entre BCs; en su lugar cada operación compleja que abarca varios BC se descompone en transacciones locales encadenadas. Así, una acción global (p.ej. crear artículo + generar precio inicial) se implementa como una saga: una secuencia de transacciones locales coordinadas. Cada paso actualiza su propia base y emite un evento que inicia el siguiente. Si falla un paso, se ejecutan transacciones compensatorias para revertir cambios previos.

Para coordinar estas sagas existen dos enfoques: **coreografía** y **orquestación**. En coreografía, cada servicio (BC) publica eventos que desencadenan acciones en otros servicios, sin controlador central. En orquestación, un servicio central (por ejemplo, el Orquestador de Apps actuando como controlador) envía comandos a participantes y gestiona el flujo. En Factura Fácil, el enfoque puede ser mixto:

- Operaciones simples pueden usar coreografía: p.ej. al crear un artículo, el Catálogo publica `ArticuloCreadoEvent` y automáticamente ListaPrecios y ControlCaja reaccionan.
- Flujos complejos o sensibles pueden usar orquestación a través del Orquestador de Apps, que invoca varios BC en orden y maneja compensaciones en caso de fallo (similar a un *Saga Orchestrator*).

Cuando se emplea coreografía, cada participante simplemente publica o escucha eventos de dominio. En el patrón orquestado, el orquestador instruye explícitamente qué hacer y cuándol. Por ejemplo, al crear una **Solicitud de producto compuesto**, el orquestador podría primero invocar el servicio de catálogo para crear el artículo base, luego llamar al servicio de precios para asignar su precio, etc. Si alguno falla, el orquestador ejecuta transacciones compensatorias (e.g. eliminar el artículo creado, reintentar o notificar error).

En ambos casos, la consistencia final se alcanza eventualmente: cada BC confirma localmente, publica su evento y el sistema convergerá a un estado consistente. Además, se pueden usar patrones complementarios:

- **Transactional Outbox:** para asegurar que la escritura en BD y publicación de evento ocurran atomically.
- **Idempotencia y Retry:** los handlers de eventos/commands deben ser idempotentes y reintentar ante fallos temporales.
- **Compensaciones:** se diseñan transacciones inversas para cada paso (p.ej. si crear artículo falla después de asignar stock, se revierte el stock).

En conclusión, la política es la de **consistencia eventual con sagas**. Los sagas pueden ser coreografiados o orquestados según la complejidad. Esto concuerda con las mejores prácticas de microservicios: “cada operación distribuida implementarse como saga... si un paso falla se ejecutan transacciones compensatorias”. El resultado es un sistema flexible que evita bloqueos globales, a costa de una complejidad mayor en coordinar y asegurar la eventual convergencia.

**Referencias:** Se utilizan patrones DDD y microservicios (Repository, Factory, Domain Events, Sagas) y mapeos de contexto para guiar este diseño. Los ejemplos y términos siguen las recomendaciones de Domain-Driven Design, aplicados al contexto de microempresas de Factura Fácil.

## Estrategia de Persistencia y Esquema de Datos\_ca

En **CatálogoArtículos** cada tenant (empresa) comparte la misma base de datos pero con particionamiento lógico. El modelo físico incluye tablas relacionales clave como **Articulo**, **Categoría**, **UnidadMedida**, etc., cada una con un campo `tenant_id` para aislar datos por empresa. Por ejemplo, la tabla `Articulo` podría definirse con columnas como:

Entidad	Campos Principales
<b>Articulo</b>	<code>id</code> (PK, UUID), <code>tenant_id</code> (FK), <code>codigo</code> (VARCHAR), <code>nombre</code> , <code>descripcion</code> , <code>precio</code> , <code>stock</code> , <code>categoria_id</code> , <code>unidad_id</code> , <code>fecha_creacion</code> , <code>fecha_actualizacion</code>
<b>Categoría</b>	<code>id</code> (PK, UUID), <code>tenant_id</code> (FK), <code>nombre</code> , <code>descripcion</code>
<b>UnidadMedida</b>	<code>id</code> (PK, UUID), <code>tenant_id</code> (FK), <code>nombre</code> , <code>simbolo</code>

Por ejemplo, un artículo podría representarse en JSON así:

```
1 {
2   "id": "e7c12a34-56f8-90ab-cdef-1234567890ab",
3   "tenantId": "empresa123",
4   "codigo": "PROD001",
5   "nombre": "Café Molido",
6   "descripcion": "Café tradicional molido",
7   "precio": 25.50,
8   "stock": 100
9 }
```

Siguiendo patrones de **DDD** y microservicios, cada servicio (bounded context) posee su propio esquema/tabla de BD privada. Así, el esquema de CatálogoArtículos es independiente y no es accesible directamente por otros servicios (solo vía API).

### 11.1 Modelo físico de entidades del catálogo ↗

- **Articulo**: incluye `id` (PK), `tenant_id`, campos de dominio (código, nombre, descripción, precio, stock, etc.). Se define `id` como clave primaria (UUID o BIGINT), y se puede usar `codigo` como clave natural única por tenant.
- **Categoría**: campos básicos de categoría (ID, tenant\_id, nombre). Relacionada con `Articulo` via `categoria_id`.
- **UnidadMedida**: si aplica, para definir unidades (e.g. kg, unidad), con su propio `tenant_id`.

Cada entidad contiene el campo `tenant_id` (FK) para particionar datos por empresa. Por ejemplo, para un **shared schema** la recomendación es incluir `tenant_id` en todas las tablas y crear un índice sobre él. Así, una tabla `Articulo` relacional podría verse SQL:

```

1 CREATE TABLE articulo (
2   id UUID PRIMARY KEY,
3   tenant_id UUID NOT NULL,
4   codigo VARCHAR(50) NOT NULL,
5   nombre VARCHAR(100),
6   descripcion TEXT,
7   precio NUMERIC(10,2),
8   stock INT,
9   categoria_id UUID,
10  unidad_id UUID,
11  fecha_creacion TIMESTAMP,
12  fecha_actualizacion TIMESTAMP,
13  UNIQUE(tenant_id, codigo) -- Código único por tenant
14 );
15 CREATE INDEX idx_articulo_tenant ON articulo(tenant_id);
16

```

- El índice compuesto `(tenant_id, codigo)` garantiza unicidad del código por empresa.
- Se indexa también `tenant_id` (y campos usados en filtros frecuentes) para acelerar consultas por tenant.

## 11.2 Índices, claves y particionamiento por tenant ☀

**Claves primarias:** Se usan IDs internos (UUID o secuenciales) como PK. Alternativamente, podría usarse clave compuesta `(tenant_id, id)`, pero es más sencillo mantener un único campo `id` con una restricción única `(tenant_id, código)` para cada producto.

### Índices recomendados:

- Índice sobre `tenant_id` en cada tabla, para particionar lógicamente los datos.
- Índices compuestos sobre campos filtrados comúnmente, p.ej. `(tenant_id, categoria_id)` o `(tenant_id, nombre)`.
- Restricción UNIQUE sobre `(tenant_id, codigo)` en `Articulo` para evitar duplicados de código por empresa.

**Particionamiento:** En bases de datos modernas (e.g. PostgreSQL, MySQL), se puede particionar horizontalmente por `tenant_id` para mejorar el rendimiento en escritorios muy grandes. Por ejemplo, se puede usar **table partitioning** por rango de `tenant_id` o un **shard** simple. Si la arquitectura lo requiere, otra opción es asignar esquemas o bases de datos separadas por tenant. (Bytebase recomienda *Database-per-Tenant* para máxima aislación, aunque implica mayor complejidad operativa; de otro modo, *Shared DB con esquema común* añadiendo `tenant_id` es viable).

## 11.3 Versionado de esquema y migraciones ☀

Para gestionar cambios de esquema se debe usar una herramienta de migraciones (p.ej. **Flyway** o **Liquibase**) con control de versiones. Se mantienen archivos de migración versionados (SQL o XML) y se ejecutan automáticamente en despliegues/CI. En entornos multi-tenant:

- Si hay un **shared schema**, solo hay un proceso de migración que ejecuta cambios en la base.
- Si se usa **schema-per-tenant**, el migrador debe aplicar cada cambio a todos los esquemas. Por ejemplo, Liquibase soporta multi-tenant con esquemas separados, aplicando el mismo changelog a cada esquema (manteniendo su propia tabla `database_changelog`).

Esto garantiza trazabilidad: cada cambio recibe un número de versión, se registra en la BD y las migraciones son idempotentes. Así se puede revertir o rehacer cambios y tener actualizaciones controladas de las tablas de catálogo.

## Seguridad y Autorización\_ca

La seguridad se delega al dominio de Identidad/Autenticación (Mi Contasis), que emite **JWT** tras validar credenciales vía OAuth2. CatalogoArticulosBC no maneja contraseñas, sino que confía en los tokens JWT de Mi Contasis. Cada petición que llega al API Gateway o servicio incluye un header `Authorization: Bearer <token>`. Spring Security (o el gateway) está configurado como *OAuth2 Resource Server*, apuntando al issuer de Mi Contasis. Por ejemplo, en Spring Boot se usaría:

```
1 spring:
2   security:
3     oauth2:
4       resourceServer:
5         jwt:
6           issuer-uri: https://mi-contasis.example.com/issuer
```

Con esto, Spring Security descarga automáticamente las claves públicas del issuer y **valida** cada JWT (firma, expiración, issuer). A nivel runtime, Spring Security comprueba la firma, tiempos `exp/nbf`, y mapea los scopes o claims a authorities internas. De este modo el servicio solo acepta peticiones con JWT válidos expedidos por Mi Contasis.

*Flujo de seguridad:* el cliente se autentica en Mi Contasis, obtiene un JWT que el API Gateway (Spring Cloud Gateway) reenvía a los microservicios. Los servicios ocultan detrás del gateway solo aceptan JWT válidos. El gateway implementa el patrón *Token Relay*, extrayendo el token del usuario y agregándolo a las solicitudes internas. Así, la autorización se basa en los claims del JWT.

### 12.1 Autenticación (JWT/OAuth2 con Mi Contasis) ↗

- **OAuth2/JWT:** Los usuarios inician sesión en Mi Contasis usando OAuth2 (p.ej. grant type `password` o `authorization_code`). Mi Contasis emite un JWT con información de usuario, roles y `tenant_id`.
- **Validez del token:** Cada servicio (o gateway) valida el JWT a cada petición. Spring Security auto-configureado con `issuer-uri` realizará la comprobación de firma, issuer y expiración.
- **Integración con Spring:** Con `spring-boot-starter-oauth2-resource-server`, basta poner el `issuer-uri` y el servicio verificará el token. Los scopes contenidos en el JWT se mapean automáticamente a authorities con prefijo `SCOPE_`, permitiendo control granular de permisos.

### 12.2 Roles y permisos requeridos ↗

Se definen roles/tipos de usuario según privilegios en el catálogo. Por ejemplo:

- **Administrador (ROLE\_ADMIN):** Permisos totales sobre el catálogo (crear/editar/eliminar artículos y categorías, gestionar importaciones).
- **Gestor/Asistente (ROLE\_EDITOR):** Puede crear y actualizar artículos, pero quizás no eliminar o cambiar configuraciones críticas.
- **Usuario/Lector (ROLE\_VIEWER):** Solo puede consultar el catálogo y listados.

Estos roles pueden expresarse en el JWT como `roles` o `scopes` (p.ej. `artcat:read`, `artcat:write`). Spring Security los interpreta como `GrantedAuthority`, permitiendo anotar endpoints con `@PreAuthorize("hasRole('ADMIN')")` o filtros equivalentes.

### 12.3 Contratos de autorización y validaciones en gateways

El **API Gateway** (p.ej. Spring Cloud Gateway o Konga/AWS API Gateway) actúa como punto único de entrada, validando JWTs antes de enrutar. En el gateway se definen contratos de autorización basados en rutas y roles: por ejemplo, rutas `POST /articulos` y `PUT /articulos/**` requieren `ROLE_ADMIN` o `ROLE_EDITOR`, mientras que `GET /articulos` está abierta a `ROLE_VIEWER` en adelante. El gateway extrae el `tenant_id` del token y lo incluye en la solicitud al microservicio, asegurando que el contexto multi-tenant se mantenga.

Además, se pueden incluir validaciones generales en el gateway (como formatos de JSON, tamaños máximos, cabeceras obligatorias, etc.) antes de alcanzar al servicio. Gracias al *token relay*, el gateway reenvía el JWT validado en las peticiones internas; así, los microservicios pueden confiar en que todo usuario llegó autenticado y autorizado según su rol.

## Monitoreo, Métricas y Alertas\_ca

Para garantizar fiabilidad se definen métricas técnicas y de negocio, usando herramientas de observabilidad (Prometheus/Grafana, etc.):

- **Métricas técnicas (RED):** Se monitorea el **Rate** de peticiones (requests por segundo), la **Error rate** (errores/segundo) y la **Duración** (latencia de respuesta) – conocido como el método *RED*. Además, métricas de infraestructura: consumo de CPU, memoria, uso de pool de conexiones, tamaño de colas/mensajes. Por ejemplo, Spring Boot Actuator con Micrometer expone métricas en formato Prometheus (HTTP servidores, JVM, etc.).
- **Métricas de negocio:** Contadores específicos del dominio, como número de artículos creados/actualizados, transacciones de importación, o número de consultas de catálogo en un periodo. Estas métricas ayudan a medir actividad de usuarios y detectar anomalías (p.ej. caídas súbitas en creación de productos).

Para visualizar y almacenar métricas se recomienda usar **Prometheus** para recolección y **Grafana** para dashboards. Por ejemplo, con Spring Boot se habilitan `micrometer-registry-prometheus` y el endpoint `/actuator/prometheus`, luego Prometheus lo scrapea regularmente. Grafana puede mostrar gráficos de latencia (p95/p99), tasa de errores, etc.

**Alertas:** Se definen en Prometheus Alertmanager reglas basadas en los SLOs. Por ejemplo, alertar si la tasa de errores de HTTP supera un umbral (p.ej. >5% por 5m), o si la latencia p95 excede cierto límite (e.g. 300ms). Según las mejores prácticas SRE, los *SLIs* clave podrían ser disponibilidad (% de respuestas OK) y latencia. Se pueden configurar SLOs internos (p.ej. 99.5% de éxito, p95 < 200ms) y alertar cuando se aproximan a violar el presupuesto de errores (error budget).

En resumen, se establecen SLIs medibles (ej. “tiempo de actividad de la API de catálogo”), SLOs asociados (por ej.  $\geq 99\%$  uptime) y SLAs formales para clientes (por ej. 99.9% uptime contractual). Atlassian define el SLI como la métrica real observada (p.ej. 99.96% uptime real) y el SLO como el objetivo prometido (p.ej. 99.95%). Las alertas de Prometheus se basarán en estos criterios, notificando a DevOps/SRE (por e-mail/Slack) si se exceden los umbrales definidos (por ejemplo, usar `alert:HighErrorRate` al detectar errores cerca del SLO).

## Estrategia de Testing\_ca

La calidad se garantiza con pruebas en varios niveles:

- **Pruebas de unidad:** Con JUnit (u otro framework Java) se testean **agregados de dominio** y servicios aislados. Por ejemplo, probar la lógica de la entidad Artículo (invariantes como precio  $\geq 0$ ), servicios de aplicación (p.ej. `CatalogoService.createArticulo()` valida reglas de negocio), y validaciones (formato de código, unicidad por inyección de un repositorio simulado). Se pueden usar *mock* (Mockito) para dependencias externas (repositorios, eventos, etc.) y asegurar cobertura de casos de borde.
- **Pruebas de contrato (Consumer-Driven Contracts):** Para la comunicación entre bounded contexts o con otros microservicios, se usan pruebas de contrato. Por ejemplo, si el Catálogo publica eventos (como `ArticuloCreado`), definimos contratos (ej. con Spring Cloud Contract) para dichos eventos. Spring Cloud Contract Verifier permite escribir el esquema de los eventos en Groovy/YAML y genera automáticamente *stubs* JSON y pruebas JUnit para validar tanto al consumidor como al productor. Esto asegura que las interfaces (HTTP APIs o mensajes) se mantengan compatibles en versiones distintas. Para eventos o mensajes JMS/Kafka se pueden usar Pact o similares.
- **Pruebas de integración/E2E:** Se levanta el microservicio completo (y sus dependencias reales simuladas o reales con Testcontainers) y se prueban flujos de extremo a extremo. Casos típicos: *crear un artículo vía la API REST*, *editar un artículo*, *listar por categoría*, *importar productos desde un archivo Excel*, etc. Estas pruebas verifican el comportamiento global: p.ej. cargamos un Excel simulado, ejecutamos el endpoint de importación y comprobamos en la base de datos que los artículos quedaron correctamente. Frameworks sugeridos: Spring Boot Test (MockMvc o WebTestClient) para API REST, TestContainers para BD en contenedor, y herramientas de UI/E2E como Selenium/Cypress si hubiera interfaz gráfica (p.ej. una web interna).
- **Herramientas:**
  - JUnit 5 (u otros) con Spring Test para unit/integ. MockMvc, @SpringBootTest, etc.
  - Mockito / AssertJ para asserts ricos en tests unitarios.
  - Spring Cloud Contract o Pact para contratos (HTTP o mensajes).
  - WireMock para stub de servicios externos en tests.
  - Testcontainers para bases de datos o colas en tests de integración.
  - Postman / RestAssured para pruebas de API manual/E2E en pipelines.
  - Cucumber (opcional) para especificaciones en Gherkin de escenarios de usuario.

# Decisiones de Arquitectura (ADR)\_ca

Este apartado documenta las principales decisiones tomadas en el diseño:

- **Event Bus vs REST síncrono:** Optamos por una arquitectura **híbrida**. Las operaciones de consulta simples (p.ej. listar o buscar artículos) se sirven síncronamente vía REST/HTTP (GET) para respuesta inmediata. En cambio, las acciones que generan cambios (crear/actualizar artículo) se publican como **eventos de dominio** en un bus de eventos (p.ej. Kafka o RabbitMQ) para lograr desacoplamiento y resiliencia. Por ejemplo, al crear un artículo emitimos `ArticuloCreado` y otros servicios (inventarios, facturación) reaccionan de forma eventual. El blog de Ambassador destaca que lo ideal es mezclar ambos patrones según la necesidad: EDA es bueno para comunicación asíncrona en tiempo real, pero REST es más adecuado para escenarios de autenticación y respuestas inmediatas. Aquí, la autenticación/autorización se hace por REST en Mi Contasis, y el catálogo publica eventos internamente para el dominio.
- **Base de datos relacional vs NoSQL:** Para el catálogo de artículos se eligió **base de datos relacional** (p.ej. PostgreSQL) por su modelo estructurado y consistencia fuerte. Cada entidad (Artículo, Categoría) tiene relaciones claras y requerimientos ACID. La documentación de Microsoft señala que las DB relacionales tienen esquemas fijos y garantizan ACID, mientras que NoSQL (documentos JSON) ofrece flexibilidad y escalabilidad en lectura pero sacrifica consistencia fuera de particiones. Dado que CatálogoArtículos requiere transacciones fiables (p.ej. asignar códigos únicos por empresa) y consultas complejas (joins por categoría), la opción relacional es la más adecuada. Si en el futuro se requieren datos muy semiestructurados (p.ej. atributos dinámicos de productos), se podría considerar un almacén NoSQL anexo, pero por ahora todo reside en una BD relacional por servicio.
- **Patrones de integración:** Evitamos transacciones distribuidas (2PC); en su lugar usamos el **patrón Saga** para procesos que abarcan múltiples servicios (ej. ajustar inventario tras crear factura). Esto permite consistencia eventual sin bloquear sistemas. Por ejemplo, si se agrega un producto al catálogo y es consumido por facturación, se puede orquestar o coreografiar un saga que compense en caso de fallo. El sitio [What are microservices?](#) recomienda usar Saga para transacciones entre servicios y CQRS (vistas materializadas por eventos) o composición de API para consultas跨servicios. En nuestro caso, las integraciones con otros Bounded Contexts (Inventario, Facturación) serán mayormente event-driven: el Catálogo emite eventos (`ArticuloActualizado`, etc.), y los consumidores mantienen sus propias vistas. Para consultas cross-context a gran escala se usaría CQRS o un enfoque *API Composition* recogiendo datos de cada servicio.
- **Herramientas de mensajería y orquestación:** Se elige **Apache Kafka** (o RabbitMQ) como bus de eventos por su alta escalabilidad y soporte de particionamiento. Con Kafka podemos particionar tópicos por `tenant_id` para preservar el orden por cliente. Por ejemplo, usando `Hash(tenant_id) % N` aseguramos que todos los eventos de una empresa vayan a la misma partición, manteniendo secuencia. Esto facilita idempotencia (guardando el `eventId` en BD con restricción única) y escalado horizontal. Como orquestador de sagas se puede usar código en un servicio dedicado o una herramienta (p.ej. Camunda); de momento se prefiere una aproximación code-driven (usando Spring Cloud Stream para productores/consumidores).
- **Resumen de decisiones clave:** Se mantiene **un esquema relacional por bounded context**, con índices y particionamiento por tenant. La comunicación es **asíncrona por eventos** cuando sea posible, cayendo a **REST síncrono** para casos que requieran respuesta inmediata (autenticación, operaciones simples). Se evita 2PC usando Saga y CQRS para consultas distribuidas. Se emplean herramientas maduras: Kafka (mensajería), Spring

Boot/Cloud (servicios), Prometheus/Grafana (monitorización) y Flyway/Liquibase (migraciones), todo alineado a los principios DDD.

# GestionClientesBC - Antonio H

## 1. Propósito

**GestionClientesBC** se encarga de administrar integralmente el ciclo de vida de los clientes en la plataforma **Factura Fácil**, asegurando que los datos maestros de clientes sean completos y consistentes. Su propósito principal es **centralizar y gestionar toda la información de clientes**, cumpliendo objetivos clave como: dar de alta clientes (manualmente vía formulario o automáticamente al emitir comprobantes), mantener un historial de versiones para auditoría, controlar la desactivación/eliminación lógica de clientes, validar la identidad de clientes con fuentes externas oficiales, gestionar múltiples contactos y adjuntos asociados, registrar el historial de operaciones (facturas, cotizaciones, pagos, incidencias) y permitir la importación masiva de clientes desde archivos.

**En resumen**, GestionClientesBC existe para resolver problemas comunes en PYMEs relacionados con la fragmentación de la información de clientes: evita errores en la facturación por datos incompletos, agiliza la atención al cliente y reduce discrepancias contables al tener un registro unificado de clientes. A continuación se detallan sus responsabilidades, contexto, interacciones con otros sistemas, modelo de dominio y otras características relevantes del Bounded Context.

### 1.1 Propósito

El propósito de **GestionClientesBC** es **administrar el ciclo de vida de los clientes** dentro de *Factura Fácil*, cubriendo desde la creación y mantenimiento de datos de clientes hasta su inactivación o eliminación. Este contexto garantiza la calidad y consistencia de los datos de clientes (nombres, documentos de identidad, contactos, etc.), sirviendo como fuente única de información de clientes para todos los demás subsistemas. Sus objetivos clave incluyen:

- **Alta de clientes:** permitir registrar nuevos clientes ya sea de forma manual (a través de formularios de registro completos) o automática (cuando se emite un comprobante de venta y el cliente no existe, se crea un cliente mínimo en línea).
- **Edición con control de versiones:** posibilitar la actualización de los datos de un cliente existente, guardando un **historial de cambios** para fines de auditoría y trazabilidad.
- **Deshabilitación y eliminación lógica:** ofrecer mecanismos para **desactivar** un cliente (pasarlo a estado inactivo, impidiendo nuevas transacciones) o **eliminarlo lógicamente** (soft-delete) cumpliendo políticas de retención de datos, asegurando consistencia referencial en el sistema.
- **Validación de identidad:** integrar con servicios externos gubernamentales (APIs de SUNAT y RENIEC) para **verificar y completar datos de identidad** (p. ej. razón social a partir de RUC, nombre a partir de DNI) y así asegurar la fidelidad de la información.
- **Gestión de contactos y adjuntos:** permitir asociar a cada cliente múltiples **medios de contacto secundarios** (emails alternos, teléfonos, direcciones) así como **documentos adjuntos** de soporte (contratos, copias de DNI, licencias, etc.) para tener un perfil de cliente completo.
- **Registro de historial de operaciones:** capturar en un historial todas las operaciones relevantes relacionadas con el cliente (emisión de facturas, recepción de pagos, cotizaciones emitidas, notas de crédito/anulaciones, incidencias de soporte, etc.), ofreciendo una **vista 360°** del cliente.
- **Importación masiva:** facilitar la carga de múltiples clientes desde archivos Excel/CSV, con validaciones de datos por fila y generación de eventos correspondientes, agilizando la incorporación inicial o actualización masiva de la cartera de clientes.

En síntesis, Gestión de Clientes BC existe para mejorar la calidad de datos, eficiencia operativa y cumplimiento normativo en la gestión de clientes. Esto aborda directamente problemas de negocio identificados: en pequeñas empresas, la falta de un registro centralizado de clientes deriva en datos duplicados o incorrectos, rechazos de comprobantes por errores en RUC/DNI, demoras en soporte al cliente y dificultad para llevar cuentas por cobrar. Gestión de Clientes BC contribuye a solucionar estos puntos mediante validaciones automáticas, interfaces unificadas y registros históricos completos.

## 1.2 Responsabilidades Clave

Las **responsabilidades principales** de Gestión de Clientes BC se derivan de su propósito. A continuación se listan las tareas y decisiones más importantes que este contexto asume, junto con una breve descripción de cada una:

Responsabilidad	Descripción
<b>Crear Cliente</b>	Registrar un nuevo cliente con todos sus datos fiscales y de contacto, incluyendo adjuntos y tipo de cliente, <b>validando formatos</b> (p. ej. de documento, email) y la <b>unicidad</b> del documento de identidad.
<b>Auto-crear Cliente desde Comprobante</b>	Al emitir un comprobante de venta con un <code>clienteId</code> desconocido, generar automáticamente un registro básico de cliente (con datos mínimos como documento y nombre) para no interrumpir la facturación. Se asegura que el cliente quede activo por defecto.
<b>Editar Cliente</b>	Permitir la modificación de los datos de un cliente existente (cambios en nombre/razón social, contactos, tipo de cliente, etc.), registrando una <b>nueva versión</b> en el historial de cambios y revalidando documentos si estos cambian.
<b>Asignar Tipo de Cliente</b>	Clasificar un cliente en una categoría de negocio (por ejemplo, <i>Mayorista</i> , <i>Minorista</i> , <i>Distribuidor</i> , etc.) para propósitos de segmentación y aplicación de precios diferenciados.
<b>Deshabilitar Cliente</b>	Cambiar el estado de un cliente a <code>INACTIVO</code> , bloqueando nuevas transacciones comerciales con ese cliente y excluyéndolo de listados operativos, mientras se conserva su información para referencias futuras.
<b>Eliminar Cliente</b>	Realizar una eliminación lógica (soft-delete) del cliente, marcándolo como eliminado una vez verificado que no existan operaciones pendientes asociadas y que se cumplan las políticas de retención de datos. La información no se borra físicamente para mantener trazabilidad.
<b>Gestionar Contacto</b>	Administrar la colección de <b>medios de contacto secundarios</b> de un cliente (por ejemplo, correo electrónico secundario, teléfono alternativo o dirección adicional). Incluye validaciones de formato según el tipo de contacto (email, número telefónico, etc.).

<b>Gestionar Adjuntos</b>	Adjuntar archivos de soporte al perfil del cliente (contratos en PDF, fotografías de documentos en JPG/PNG, etc.) o eliminarlos. Se valida el tipo de archivo permitido y que el tamaño no exceda el máximo (p. ej. 5 MB).
<b>Validar Identidad Externa</b>	Invocar servicios externos de identidad (API de SUNAT para RUC, API de RENIEC para DNI) para <b>completar y verificar</b> los datos ingresados del cliente (razón social, nombre legal, dirección fiscal), garantizando su validez.
<b>Registrar Operación en Historial</b>	Capturar eventos de otros contextos (por ejemplo, <b>ComprobanteEmitido</b> o <b>PagoRecibido</b> ) y generar un registro histórico de la operación en el perfil del cliente (por ejemplo, agregar un elemento que indique que se emitió una factura de cierto monto).
<b>Consultar Historial de Operaciones</b>	Permitir la consulta filtrada del historial de operaciones de un cliente: listar facturas, pagos u otros eventos asociados, filtrados por rango de fechas, tipo de operación o estado.
<b>Importar Clientes desde Excel/CSV</b>	Procesar un archivo Excel o CSV con una lista de clientes para carga masiva. El sistema valida cada fila (formato de campos, duplicados, etc.) y crea o actualiza los registros correspondientes, generando eventos <b>ClienteCreado</b> o <b>ClienteModificado</b> .
<b>Consultar Cliente</b>	Obtener la “ficha” completa de un cliente, incluyendo datos principales, contactos, adjuntos e incluso un resumen de su historial de operaciones recientes, para mostrarlo en pantalla o integrarlo en otros procesos.

### 1.3 Contexto y Motivación ☰

En el ámbito de Factura Fácil (dirigido a pequeñas y microempresas), **GestionClientesBC** surge para resolver problemas comunes en la gestión de clientes. En muchas PYMEs, la información de clientes suele estar dispersa o incompleta, lo que provoca errores en la facturación (por ejemplo, RUC mal escritos que derivan en rechazos por SUNAT), retrasos en el soporte al cliente por falta de datos de contacto actualizados, y discrepancias contables por registros duplicados o inconsistentes. A continuación se resumen las motivaciones clave y cómo este contexto las aborda:

- **Calidad de datos:** Implementa **validaciones automáticas** (formato correcto de DNI/RUC, correo electrónico válido, etc.) y flujos de verificación manual cuando es necesario. Esto reduce significativamente los rechazos de comprobantes electrónicos por datos erróneos, asegurando que la información enviada a SUNAT esté correcta.
- **Eficiencia operativa:** Proporciona formularios completos y funcionalidades como la **carga masiva de clientes** desde Excel, lo que minimiza el tiempo invertido en el ingreso de datos. Los usuarios pueden registrar o actualizar decenas de clientes rápidamente en lugar de hacerlo uno por uno.
- **Visibilidad 360° del cliente:** Consolida en un solo lugar todo el historial de interacciones y transacciones con cada cliente (facturas emitidas, pagos recibidos, cotizaciones, etc.), facilitando la gestión de cuentas por cobrar y permitiendo una atención más personalizada y oportuna.

- **Cumplimiento y auditoría:** Mediante la conservación de versiones históricas de los datos del cliente, el registro de eventos importantes (creación, modificación, deshabilitación) y la aplicación de eliminaciones lógicas (soft-delete en vez de borrados físicos), se garantiza la **trazabilidad** de cada cambio y el cumplimiento de normativas (tributarias y de protección de datos). Esto brinda confianza en que existe un rastro auditabile de las operaciones relacionadas con clientes.

En resumen, GestiónClientesBC existe porque **un registro maestro unificado de clientes** es fundamental para la operación consistente de Factura Fácil. Sin este contexto, habría mayor riesgo de errores fiscales, operaciones duplicadas y dificultades para escalar el sistema a más empresas y transacciones.

#### 1.4 Fuentes de Información e Integraciones

El diseño e implementación de GestiónClientesBC se nutre de diversas **fuentes de información** y se integra con múltiples sistemas internos y externos. Esto incluye tanto lineamientos externos (normativas fiscales para validación de RUC/DNI) como la colaboración con otros Bounded Contexts dentro de la solución. A continuación se detallan las principales integraciones y fuentes que influyen en GestiónClientesBC:

- **ComprobantesElectronicosBC (Módulo de Facturación Electrónica):** *Inbound* – GestiónClientesBC **recibe eventos** desde el contexto de facturación, tales como `ComprobanteEmitido` y `ComprobanteAnulado`, los cuales incluyen el `clienteId` y datos relevantes (tipo de comprobante, montos, etc.). Estos eventos disparan lógica en GestiónClientesBC para auto-crear clientes desconocidos o registrar operaciones en el historial del cliente. (*Ejemplo: al recibir un evento de factura emitida con un cliente no registrado, se invoca el caso de uso de auto-creación de cliente.*)
- **ControlCajaBC (Módulo de Control de Caja/Pagos):** *Inbound* – GestiónClientesBC **consume el evento PagoExternoRecibido** proveniente del contexto de caja. Cuando se registra un pago en efectivo u otro canal externo asociado a un cliente, este evento permite crear una entrada de tipo “pago recibido” en el historial del cliente correspondiente.
- **Servicio de Identidad (APIs Externas SUNAT y RENIEC):** *Outbound/External* – GestiónClientesBC se integra con servicios gubernamentales de identidad para **consultar y validar datos fiscales y personales**. En particular, consulta la API de SUNAT para verificar la validez de un RUC y obtener la razón social y dirección fiscal de empresas, y la API de RENIEC para validar números de DNI y obtener nombres completos. Estas fuentes externas aseguran que los datos de identidad ingresados coincidan con registros oficiales, mejorando la calidad de datos.
- **IndicadoresNegocioBC (Módulo de Métricas y Reportes):** *Outbound* – GestiónClientesBC **publica eventos de dominio** que son consumidos por el contexto de métricas. Por ejemplo, eventos como `ClienteCreado`, `ClienteModificado`, `ClienteDeshabilitado` u `OperacionClienteRegistrada` son enviados para que IndicadoresNegocioBC actualice dashboards, KPI y reportes en tiempo real. (*Ejemplo: al crearse un cliente nuevo, IndicadoresNegocioBC podría incrementar el contador de clientes activos en un panel de control.*)
- **XlsImportService (Servicio/Adapter de Importación):** *Inbound* – GestiónClientesBC se apoya en un servicio de importación de archivos Excel/CSV para la funcionalidad de carga masiva. Este servicio externo/proveedor entrega a GestiónClientesBC el contenido parseado de un archivo de clientes que el usuario sube, fila por fila. GestiónClientesBC procesa esos datos invocando las operaciones de creación/actualización correspondientes por cada registro, generando eventos por cada fila procesada. (*Nota: XlsImportService actúa más como un adaptador que como un microservicio independiente; podría estar implementado dentro del mismo contexto como componente de aplicación.*)
- **Otros:** Adicionalmente, GestiónClientesBC toma en cuenta **normativas tributarias y comerciales locales** como fuentes de requerimientos. Por ejemplo, las reglas de validación de RUC (11 dígitos con algoritmo de dígito verificador) y de DNI (8 dígitos numéricos) provienen de estándares definidos por SUNAT y RENIEC

respectivamente. Asimismo, la validación de correo electrónico se basa en el estándar **RFC 5322** (formato válido de direcciones de email). Estas normas externas han influido directamente en las reglas de negocio implementadas en el contexto (ver sección 5).

**Nota:** GestiónClientesBC está diseñado para operar en un entorno **multi-tenant** y **multi-empresa**. Esto significa que los datos de clientes están particionados por compañía o tenant dentro de la plataforma Factura Fácil (p. ej. un mismo identificador de cliente podría existir en dos empresas distintas sin colisión). La autenticación y gestión de tenants se realiza a nivel de un **Contexto de Identidad y Autenticación** separado (denominado "Mi Contasis"), mediado por un orquestador de aplicaciones. Si bien esto afecta cómo GestiónClientesBC filtra y accede a datos, la lógica específica de autenticación/autorización se detalla en secciones posteriores (ver sección 12) dado que reside fuera de este contexto.

## 1.5 Modelo de Dominio

El modelo de dominio de GestiónClientesBC se centra en el **agregado Cliente** como entidad raíz, alrededor del cual se agrupan varias entidades internas y objetos de valor para mantener las invariantes del negocio. A alto nivel, un **Cliente** representa a una persona o empresa que interactúa con Factura Fácil (como receptor de facturas, pagador, etc.), y encapsula dentro de sí todos los datos y comportamientos relevantes: sus datos maestros (identificación y clasificación), sus medios de contacto, sus documentos adjuntos y su historial de operaciones.

### 1.5.1 Agregado Raíz y Entidades del Agregado

- **Cliente (Agregado Raíz):** Es la **entidad raíz** del contexto, que orquesta las sub-entidades y objetos de valor relacionados con un cliente. Un objeto Cliente contiene los **datos maestros** (por ejemplo, nombre/razón social, documento de identidad, tipo de cliente, estado, preferencias por defecto de moneda y forma de pago), así como referencias a colecciones de:
  - **Contactos (ContactoCliente)** – medios de contacto secundarios del cliente (emails alternativos, teléfonos, direcciones).
  - **Adjuntos (AdjuntoCliente)** – archivos/documentos asociados al cliente.
  - **Operaciones (OperacionCliente)** – registros históricos de operaciones/transacciones relacionadas con el cliente.

El agregado Cliente **garantiza invariantes** importantes, como que no se duplique el documento de identidad y que sus listas internas (contactos, adjuntos, operaciones) se mantengan consistentes. Posee métodos para las principales operaciones de negocio: crear un cliente, editar sus datos, deshabilitarlo, eliminarlo lógicamente, registrar una nueva operación, etc.. Todas estas acciones pasan por la entidad raíz para asegurar la consistencia del agregado.

**Relaciones:** Un Cliente puede tener *uno a muchos* ContactoCliente, *uno a muchos* AdjuntoCliente y *uno a muchos* OperacionCliente vinculados a él. Cada sub-entidad pertenece exclusivamente a un Cliente (es decir, no existen contactos compartidos entre clientes, por ejemplo).

- **ContactoCliente (Entidad interna del agregado Cliente):** Representa un **medio de contacto adicional** de un cliente. Por ejemplo, un cliente podría tener un correo electrónico secundario, uno o más números telefónicos alternativos, y/o direcciones postales adicionales registrados bajo esta entidad. Cada ContactoCliente generalmente registra el tipo de contacto (por ejemplo, *EMAIL\_SECUNDARIO*, *TELEFONO\_SECUNDARIO* o *DIRECCION*) y el valor (la dirección de email, el número telefónico, etc.).
  - **Comportamiento:** El agregado Cliente provee operaciones para **agregar, editar o eliminar contactos** (`agregarContacto()`, `editarContacto()`, `eliminarContacto()`), aplicando validaciones de

formato según el tipo (por ejemplo, validar sintaxis de email). Al agregar o quitar un contacto, se generan los eventos de dominio correspondientes (ver eventos **ContactoAgregado** y **ClienteModificado** más adelante).

- **AdjuntoCliente (Entidad interna del agregado Cliente):** Corresponde a un **documento o imagen adjunta** al perfil de un cliente. Esta entidad almacena metadatos del archivo (tipo de adjunto, nombre de archivo, ruta o URL de almacenamiento, comentario, etc.) y posiblemente el enlace al contenido almacenado (por ejemplo en un sistema de archivos o S3). Los adjuntos comunes incluyen contratos firmados, copias de documentos de identidad, licencias, fotos del cliente o cualquier otro soporte documental.
  - *Comportamiento:* El agregado Cliente permite **agregar y eliminar adjuntos** (`agregarAdjunto()`, `eliminarAdjunto()`), verificando que el tipo de archivo y tamaño sean válidos antes de adjuntar. La eliminación suele ser lógica (marcando el adjunto como eliminado o moviéndolo a un historial) para mantener registro. Al agregar un adjunto nuevo se produce el evento **AdjuntoAgregado**, y al eliminar se considera un **ClienteModificado** (ya que es un cambio en la ficha del cliente).
- **OperacionCliente (Entidad interna del agregado Cliente):** Esta entidad registra una **transacción u operación histórica relevante** asociada al cliente. Cada OperacionCliente podría representar por ejemplo: una factura emitida al cliente, una cotización, un pago recibido de él, una nota de crédito/anulación de factura, entre otros. Los campos clave de OperacionCliente incluyen:
  - `tipoOperacion` – el tipo de operación (ver **TipoOperacion** en objetos de valor),
  - `montoOperacion` – el importe monetario involucrado (ver **MontoOperacion**),
  - `fechaOperacion` – la fecha/hora en que ocurrió la operación,
  - `referenciaId` – un identificador que vincula esta operación con la entidad externa correspondiente (por ejemplo, el ID del comprobante emitido o del pago registrado).
  - `operacionId` – identificador interno de la operación en el contexto de cliente (UUID).

*Comportamiento:* Normalmente, las OperacionCliente **no se manipulan directamente por el usuario**, sino que se crean automáticamente al recibir eventos de otros BC. El agregado Cliente tiene un método `registrarOperacion()` para agregar una nueva operación a su historial, asegurando que:

- La fecha de la operación no sea anterior a la fecha de registro del cliente (regla de integridad temporal, ver RN-GC-005 en sección 5).
- Se actualicen los saldos o métricas internas si correspondiera (por ejemplo, podría haber un cálculo de total facturado, total pagado, etc., aunque esas métricas globales normalmente las calcula IndicadoresNegocioBC).

Cuando se registra una OperacionCliente, se genera el evento **OperacionClienteRegistrada** para notificar a otros contextos (indicadores, etc.). Esta entidad sirve principalmente como *registro inmutable* dentro del agregado; típicamente no se edita ni elimina una vez creada (más allá de posibles correcciones manuales en caso de errores, lo que estaría sujeto a reglas estrictas).

### 1.5.2 Objetos de Valor (Value Objects)

El contexto GestionClientesBC utiliza varios **Value Objects (VO)** para encapsular conceptos y reglas de negocio inmutables, garantizando la consistencia en el modelo. Los VO son comparados por su valor y ayudan a validar información crítica antes de persistirla. A continuación se describen los principales objetos de valor del dominio:

- **DocumentoIdentidad:** Objeto de valor que combina el tipo de documento y el número de documento de un cliente, encapsulando la lógica de **validación de formato y dígito verificador** cuando aplique. Un

DocumentoIdentidad garantiza, por ejemplo, que si el tipo es RUC, el número tenga 11 dígitos numéricos y pase la validación del dígito de control, o que si es DNI, sean 8 dígitos. Este VO se usa como identificador único de clientes (dos clientes no pueden compartir el mismo DocumentoIdentidad) y es inmutable una vez creado. Representa formalmente la identidad fiscal o personal del cliente en el sistema.

- **TipoDocumento:** Es un enumerado (enum) que indica el **tipo de documento de identidad** utilizado. Los valores típicos incluyen *DNI*, *RUC*, *Pasaporte*, *Carné de Extranjería*, *Licencia de conducir*, etc. Cada TipoDocumento define indirectamente las reglas de validación para el número de documento (longitud, caracteres permitidos, existencia de dígito verificador). Este enum es usado en formularios y validaciones para restringir entradas y guiar la lógica (por ejemplo, solo si el tipo es RUC se consulta la API de SUNAT). **Ejemplo:** TipoDocumento = RUC implica 11 dígitos numéricos y verificación con algoritmo de SUNAT.
- **TipoCliente:** Enumeración que clasifica al cliente según su categoría comercial. Valores posibles: *MAYORISTA*, *MINORISTA*, *DISTRIBUIDOR*, *REVENDEDOR*, etc. Este VO influye en reglas de negocio como la aplicación de **listas de precios** segmentadas o condiciones comerciales especiales para el cliente. Por ejemplo, un cliente tipo DISTRIBUIDOR podría tener descuentos especiales. El TipoCliente es asignado al crear o editar un cliente y puede cambiarse en cualquier momento (p. ej. reclasificar un cliente de minorista a mayorista) – generando un evento ClienteModificado.
- **EstadoCliente:** Enumeración del estado del cliente. Valores: *ACTIVO* o *INACTIVO* (no se menciona explícitamente en doc un estado *ELIMINADO*, dado que la eliminación es lógica, pero internamente podría representarse con un estado adicional o un flag separado). El EstadoCliente *controla la disponibilidad* del cliente para operaciones: solo clientes ACTIVO pueden ser usados en nuevas transacciones, mientras que clientes INACTIVO aparecen bloqueados (intentos de facturarle deberían rechazarse, etc.). Cuando se auto-crea un cliente, por regla de negocio se asigna siempre ACTIVO. Deshabilitar un cliente consiste precisamente en cambiar este estado a INACTIVO.
- **FechaRegistro:** Un objeto (o simplemente un campo de valor) que guarda la **fecha y hora de creación o de última modificación** significativa del cliente. Suele almacenarse en formato de *timestamp* estándar (por ejemplo ISO 8601). Se utiliza para fines de auditoría y ordenamiento cronológico. Cada vez que se realiza una operación como editar datos de cliente o agregar un adjunto, se actualiza el FechaRegistro del cliente. Sirve también en reglas como RN-GC-005 (una operación registrada no debe tener fecha anterior a la FechaRegistro del cliente).
- **TipoOperacion:** Enumeración usada dentro de OperacionCliente para clasificar el tipo de operación histórica registrada. Posibles valores: *FACTURA\_EMITIDA*, *COTIZACION\_EMITIDA*, *PAGO\_RECIBIDO*, *FACTURA\_ANULADA*, etc. Este VO permite que el historial distinga la naturaleza de cada registro y que se apliquen lógicas específicas según el tipo (por ejemplo, solo las FACTURA\_EMITIDA suman al total facturado, los PAGO\_RECIBIDO afectan saldo, etc.). También puede determinar cómo se muestran las operaciones en reportes.
- **MontoOperacion:** Objeto de valor para representar un **importe monetario** asociado a una operación. Encapsula un decimal con la precisión adecuada (por ej. dos decimales para moneda local) y posiblemente la moneda si se requiere. Este VO asegura que cálculos de saldo y métricas se hagan con una representación consistente del dinero. Podría incluir validaciones como no permitir montos negativos inesperados, etc.
- **ReferenciaId:** Identificador de referencia externo (generalmente un UUID o string) que vincula una OperacionCliente con la entidad original en otro contexto. Por ejemplo, si TipoOperacion = FACTURA\_EMITIDA, el ReferenciaId podría ser el ID (UUID) del comprobante electrónico emitido en ComprobantesElectronicosBC; si es PAGO\_RECIBIDO, podría ser el ID del registro de pago en ControlCajaBC. Este VO no tiene lógica interna más allá de almacenar el identificador, pero es crucial para correlacionar los eventos entre contextos y permitir navegabilidad cruzada (por ej., desde el historial del cliente poder abrir el comprobante correspondiente).

Todos los objetos de valor mencionados son **inmutables** y se comparan por igualdad de sus atributos. Esto significa que cualquier cambio (por ejemplo, corregir un número de RUC mal ingresado) implica crear un nuevo VO

(DocumentoIdentidad) en lugar de modificar el existente, garantizando así que las versiones antiguas permanezcan intactas para el historial.

### 1.5.3 Servicios de Dominio ☈

Dentro de GestiónClientesBC se identifica al menos un **Servicio de Dominio** importante, utilizado para coordinar lógicas de negocio que trascienden una sola entidad:

- **ValidacionIdentidad:** Es un servicio de dominio cuya responsabilidad es **consultar las APIs externas de identidad (SUNAT y RENIEC)** para verificar y complementar los datos de un cliente. Este servicio encapsula la lógica de comunicación con estos sistemas externos y las reglas de decisión sobre cómo integrar los datos obtenidos. Por ejemplo, al crear o editar un cliente, ValidacionIdentidad podría exponer un método `validarCliente(cliente)` que:
  - a. Según el tipo de documento del cliente (p. ej. RUC), llama a la API correspondiente (SUNAT) enviando el número de documento.
  - b. Recibe la información oficial (razón social, dirección fiscal si es RUC; nombres completos si es DNI).
  - c. Compara o reemplaza datos en el agregado Cliente (si están vacíos los completa, si están llenos podría verificarlos).
  - d. Devuelve el resultado de la validación (ej. éxito, documento no existente, servicio no disponible, etc.).

Integrar esta lógica en un servicio de dominio tiene sentido porque **no pertenece a ninguna entidad en particular**, sino que es una operación transversal que enriquece la entidad Cliente. Además, en términos de orquestación, este servicio podría ser utilizado tanto en el caso de uso de *Crear Cliente* como en un comando explícito de *Validar Identidad Externa*.

- **(Otros Servicios de Dominio):** Fueras de ValidacionIdentidad, la mayor parte de la lógica de negocio recae en el propio agregado Cliente. No obstante, podrían concebirse otros servicios de dominio si fuese necesario. Por ejemplo, si la importación masiva requiriera lógica compleja compartida (como transformación de datos), se podría tener un servicio de dominio *ImportacionClientesService* que procese las filas, aunque en la implementación actual actúa más bien como servicio de aplicación/orquestación (ver sección 7). En resumen, cualquier lógica compleja que **no encaje naturalmente en una sola entidad** se delega a servicios de dominio para mantener el modelo limpio, pero en GestiónClientesBC el único destacado explícitamente es ValidacionIdentidad.

### 1.6 Relaciones con otros Bounded Contexts ☈

GestiónClientesBC se relaciona con varios otros **Bounded Contexts (BC)** dentro de la solución Factura Fácil, principalmente mediante **contratos de eventos** publicados o consumidos. A continuación se describen dichas relaciones con cada contexto relevante, indicando el tipo de vínculo y la información compartida:

- **CatalogoArticulosBC:** *Relación: Ninguna integración directa* identificada. GestiónClientesBC y CatalogoArticulosBC (que gestiona el catálogo de productos/servicios) operan en dominios distintos. No hay eventos ni datos compartidos directamente, ya que la información de clientes no interfiere con la de artículos. (*Cada uno mantiene sus propias entidades: clientes por un lado, productos por otro.*)
- **ComprobantesElectonicosBC (Facturación Electrónica):** *Relación: Cliente-Proveedor (Cliente)* – Desde la perspectiva de Facturación, GestiónClientesBC actúa como **proveedor de datos maestros de cliente**. Cuando el módulo de Comprobantes necesita datos del cliente (por ejemplo, para poblar la factura con razón social y dirección, o validar si un cliente está activo), consulta a GestiónClientesBC.
- **Eventos Compartidos:** GestiónClientesBC **consume eventos inbound** producidos por ComprobantesElectonicosBC:

- **ComprobanteEmitido** – cuando se emite una factura/boleta electrónica, este evento (que incluye `clienteId`, totales, forma de pago, moneda, etc.) es recibido por GestionClientesBC. Si el `clienteId` no existe en su registro, dispara el caso de uso *Auto-crear Cliente*. Siempre que recibe un ComprobanteEmitido, registra una OperacionCliente de tipo FACTURA\_EMITIDA en el historial del cliente correspondiente.
- **ComprobanteAnulado** – cuando se anula un comprobante previamente emitido, GestionClientesBC lo recibe y agrega una OperacionCliente de tipo FACTURA\_ANULADA, reflejando la anulación en el historial.

**Consumidor de Eventos de Cliente:** A su vez, ComprobantesElectonicosBC **podría suscribirse** a ciertos eventos de clientes. Por ejemplo, se espera que consuma el evento `ClienteDeshabilitado` para, llegado el caso, **bloquear la emisión de nuevas facturas** a ese cliente desde el módulo de facturación (evitando así ventas a clientes inactivos). Este vínculo muestra cómo ComprobantesBC confía en la información de estado de clientes provista por GestionClientesBC.

- **ControlCajaBC (Gestión de Pagos/Caja): Relación: Integración mediante eventos** – GestionClientesBC **consume** el evento `PagoExternoRecibido` publicado por ControlCajaBC. Este evento contiene datos de un pago registrado en caja (monto, fecha, `clienteId`, etc.). Al capturarlo, GestionClientesBC inserta un registro OperacionCliente de tipo PAGO\_RECIBIDO en el historial del cliente, actualizando así su situación de cuentas por cobrar.

**Possible consumo inverso:** Si un cliente es deshabilitado o eliminado, ControlCajaBC podría opcionalmente suscribirse a esos eventos para evitar aceptar pagos de clientes inactivos; sin embargo, el caso más claro es que la lógica de negocio reside en Facturación (no emitir facturas a inactivos). En cualquier caso, no se detallan eventos hacia ControlCajaBC en la documentación disponible, lo que sugiere una relación principalmente de entrada hacia GestionClientesBC.

- **ListaPreciosBC (Gestión de Listas de Precios): Relación: Integración mediante eventos de cliente** – ListaPreciosBC se encarga de precios segmentados por cliente, canal, etc. Este contexto **consume eventos publicados** por GestionClientesBC relacionados a cambios en clientes. En particular, cuando ocurre un evento `ClienteModificado` (por ejemplo, cambio de `TipoCliente` o estado) en GestionClientesBC, ListaPreciosBC lo recibe para **actualizar la segmentación o reglas de precios asociadas a ese cliente**. Por ejemplo, si un cliente cambia de categoría de *MINORISTA* a *MAYORISTA*, ListaPreciosBC recalculará qué lista de precios le corresponde.

Además, si un cliente es deshabilitado, ListaPreciosBC podría marcar que ya no apliquen ciertos precios específicos asociados. Esta relación es típicamente *Publisher/Subscriber*, con GestionClientesBC publicando y ListaPreciosBC reaccionando (tipo de relación *Customer-Supplier* inverso: ListaPrecios es *cliente* de la info de cliente para sus procesos).

- **IndicadoresNegocioBC (Dashboards/Métricas): Relación: Publisher/Subscriber (Eventual Consistency)** – GestionClientesBC es un productor de eventos que **alimentan los dashboards y reportes** manejados por IndicadoresNegocioBC. Todos los eventos de dominio de clientes son consumidos por IndicadoresNegocioBC para fines de análisis:

- **ClienteCreado** – incluir al nuevo cliente en las métricas (p. ej., conteo de clientes activos, segmentación por tipo).
- **ClienteModificado** (o `ClienteActualizado` en la terminología de métricas) – refrescar datos de ese cliente en los reportes y generar alertas si aplica (p. ej., notificar al área de ventas si cambió de segmento).

- **ClienteDeshabilitado** – excluir al cliente de cálculos futuros (ya no se cuenta en clientes activos, no aporta a ventas proyectadas).
- **OperacionClienteRegistrada** – cuando se registra una nueva operación (venta o pago) se actualizan los totales de ventas, cobranzas, etc., en los dashboards financieros.

La relación es *open-host service*, donde GestiónClientesBC publica eventos y no conoce a sus consumidores; IndicadoresNegocioBC se suscribe para reaccionar. Esto logra un bajo acoplamiento y consistencia eventual en las métricas.

- **ConfiguracionSistemaBC (Parámetros Globales): Relación: Conformist/Shared Kernel** – Aunque no se menciona un vínculo directo en eventos, GestiónClientesBC probablemente **consume datos estáticos** o parámetros desde ConfiguracionSistemaBC para ciertas funcionalidades. Por ejemplo, al registrar un cliente se pide *monedaPredeterminada* y *formaPagoPredeterminada*, las cuales podrían provenir de catálogos globales definidos en ConfiguracionSistemaBC (lista de monedas soportadas, formas de pago estándar). Asimismo, los *tipos de documento* podrían ser una lista administrada centralmente.

En la documentación general, ConfiguracionSistemaBC provee parámetros como moneda por defecto del sistema, formatos, etc. que impactan a múltiples BC. En el caso de GestiónClientesBC:

- La **moneda por defecto** podría pre-llenarse desde ConfiguracionSistemaBC cuando se crea un cliente (ej. PEN o USD según configuración global).
- Las **formas de pago** aceptadas (contado, crédito 30 días, etc.) posiblemente se definen globalmente y GestiónClientesBC solo las referencia.

Esta relación se asemeja a *Conformist*, donde GestiónClientesBC toma los catálogos tal cual los define ConfiguracionSistemaBC sin traducirlos (p. ej. usa los códigos de moneda, unidades, etc. proporcionados).

- **Identidad y Autenticación (Mi Contasis, Orquestador): Relación: Contexto separado (ACL para tenencia)** – Si bien estrictamente es de otro dominio (no de Factura Fácil sino transversal), cabe mencionar que GestiónClientesBC opera en un entorno multi-tenant donde un **Orquestador de Apps** y un BC de Identidad gestionan qué usuarios/empresas tienen acceso a qué datos de clientes. La interacción aquí no es directa a nivel de entidad de cliente, sino a nivel de seguridad: los APIs de GestiónClientesBC reciben un **JWT con el tenant (empresa)** y un usuario, y gracias a eso filtran y aseguran que un usuario solo gestione clientes de su empresa. Este detalle de integración con identidad y control de acceso se amplía en la sección de Seguridad (sección 12).

En conclusión, GestiónClientesBC juega un rol central sirviendo como **fuente de verdad de datos de clientes** para otros contextos (facturación, precios, métricas) y a la vez escucha eventos de facturación y pagos para mantener actualizado el historial de cada cliente. Los patrones principales en estas relaciones son *Customer-Supplier* (p. ej. Facturación depende de datos de cliente), *Publisher-Subscriber* (eventos de cliente hacia métricas, listas de precio) y uso de *anticorruption layers* simples para integrar con APIs externas (SUNAT/RENIEC).



## 1.7 Restricciones y Reglas del Negocio

GestiónClientesBC define una serie de **invariantes y políticas de negocio** que **siempre deben cumplirse** para garantizar la integridad del sistema y adherirse a normativas. A continuación se enumeran las principales restricciones (condiciones que impiden realizar una operación si no se cumplen) y reglas de negocio (condiciones lógicas que dictan comportamientos dentro de un flujo) del contexto de clientes:

- **Documento de Identidad único y válido:** Un mismo número de documento (DNI, RUC, etc.) **no puede registrarse en dos clientes distintos**, y debe cumplir el formato específico de su tipo. Por ejemplo, un RUC debe tener 11

dígitos numéricos válidos y dígito verificador correcto, un DNI debe tener 8 dígitos, etc. *Referencias:* Esta restricción garantiza la unicidad del cliente en la base de datos y evita conflictos legales (RN-GC-001).

- **Datos obligatorios del cliente:** Todo cliente **debe tener al menos un nombre/razón social y un documento de identidad válido** registrado. En otras palabras, no se permiten clientes anónimos ni sin identificación fiscal (RN-GC-001, regla de negocio). Si falta alguno de estos campos al intentar crear un cliente, la operación es rechazada.
- **Formato de correo electrónico:** Si se registra un email para el cliente (sea principal o secundario), **debe cumplir el formato estándar RFC-5322** (por ejemplo, `usuario@dominio.com`). Un correo mal formado hará fallar la validación (RN-GC-002).
- **Estado INACTIVO bloquea operaciones:** Un cliente marcado como **INACTIVO** no puede ser usado en nuevas transacciones (no se le pueden emitir facturas, generar pedidos, etc.). Cualquier intento de incluir un cliente inactivo en una factura debe ser impedido por las validaciones del sistema o por los módulos consumidores (RN-GC-003).
- **Auto-creación en estado ACTIVO:** Cuando el sistema **auto-crea un cliente** a partir de un evento de comprobante (es decir, un cliente que no existía y se genera “al vuelo”), este debe crearse siempre con estado ACTIVO. La lógica define RN-GC-004: asignar `EstadoCliente = ACTIVO` por defecto en ese escenario, para que la factura pueda continuar procesándose y el cliente quede habilitado.
- **Registro temporal de operaciones:** Cada registro de historial (`OperacionCliente`) debe tener su `fechaOperacion posterior o igual a la fecha de registro del cliente` en sí. Es decir, no puede haber una operación fechada antes de que el cliente exista (RN-GC-005). Esta regla asegura consistencia cronológica en la data.
- **Eliminación de cliente condicionada:** Para **eliminar lógicamente un cliente**, se debe verificar que **no tenga operaciones pendientes** (por ejemplo, facturas impagadas) u otros vínculos críticos activos. Solo cumpliendo esto, se procede a marcarlo como eliminado (soft-delete). Además, se deben cumplir políticas de retención de datos (por ejemplo, no eliminar clientes con movimientos en el último X tiempo si así lo exige una norma). Esta política se referencia como RN-GC-006.
- **Restricciones sobre adjuntos:** Los archivos adjuntos que se suben a un cliente deben ser de tipo permitido y tamaño limitado. En concreto, **solo se aceptan archivos PDF, JPG o PNG de hasta 5 MB** de tamaño. Cualquier adjunto que exceda este límite o no corresponda a esos formatos debe ser rechazado en el momento de la carga.

Estas reglas y restricciones están respaldadas tanto por validaciones en el modelo de dominio (por ejemplo, la clase `DocumentoIdentidad` valida formato en su constructor), como por comprobaciones en los flujos de casos de uso (p. ej. antes de deshabilitar o eliminar un cliente se realizan las verificaciones correspondientes). En la siguiente tabla se resumen algunas reglas con sus códigos y descripciones breves:

Código	Regla / Restricción
RN-GC-001	El <b>DocumentoIdentidad</b> de un cliente debe ser único en el sistema y cumplir el formato específico según su <i>TipoDocumento</i> (longitud y dígito verificador). Además, tanto el <code>DocumentoIdentidad</code> como el <b>Nombre o Razón Social</b> son obligatorios para dar de alta un cliente (no se permite ninguno de los dos en blanco)
RN-GC-002	El <b>correo electrónico</b> registrado en un cliente debe tener un formato válido de acuerdo al estándar RFC-5322 (ejemplo: <code>usuario@dominio.com</code> ). Un email inválido provoca error de validación.

RN-GC-003	Un cliente en estado <b>INACTIVO</b> no puede realizar nuevas transacciones ni ser destinatario de comprobantes. Esta es una restricción operacional: antes de facturar o registrar un pago, se verifica que el cliente esté <b>ACTIVO</b> .
RN-GC-004	Al <b>auto-crear</b> un cliente (vía evento de comprobante), el cliente se debe crear siempre en estado <b>ACTIVO</b> . Esto asegura que la operación que motivó su creación (ejemplo: emisión de factura) pueda continuar sin bloqueos.
RN-GC-005	Cada entrada de <b>OperacionCliente</b> registrada debe tener una <b>fechaOperacion</b> mayor o igual a la <b>FechaRegistro</b> del cliente. Nunca se debe registrar en el historial una operación con fecha anterior a la creación del cliente.
RN-GC-006	Para <b>eliminar</b> lógicamente un cliente, se debe verificar la <b>ausencia de operaciones pendientes</b> asociadas (p. ej., facturas no cobradas). Solo tras esa verificación, se procede con el soft-delete del cliente.
Adjuntos	Los <b>Adjuntos</b> de un cliente deben ser archivos de tipo permitido (PDF, JPG o PNG) y con tamaño ≤ 5 MB <a href="#">loaicite:97</a> . Si un archivo no cumple estas condiciones, no se permite adjuntarlo al cliente.

**Cumplimiento de normativas:** Varias de estas reglas están motivadas por cumplir normativas externas. Por ejemplo, la unicidad del RUC y su formato correcto está alineada con exigencias de SUNAT (evitando errores en libros electrónicos y facturación electrónica). La restricción de no eliminar clientes con operaciones pendientes responde a prácticas contables y de auditoría (no eliminar registros que puedan ser necesarios para justificar transacciones financieras). Asimismo, las validaciones de email y tamaño de archivos contribuyen a la calidad de datos y a evitar problemas técnicos (un email inválido impediría contactar al cliente; archivos demasiado grandes afectarían el almacenamiento o desempeño).

En la implementación, estas reglas se aplican en diferentes momentos: algunas **antes de ejecutar un caso de uso** (restricciones iniciales, p. ej. “no duplicar documento” se chequea al inicio de Crear Cliente), otras **durante el flujo** (p. ej. “asignar ACTIVO en auto-creación” ocurre en medio del proceso), y otras al final para garantizar consistencia (p. ej. publicar eventos solo si todo está correcto). Todas juntas conforman la *política de negocio* de GestionClientesBC.

## 1.8 Valor Estratégico

El Bounded Context GestionClientesBC aporta un **valor estratégico significativo** tanto a la solución Factura Fácil en su conjunto, como al negocio del cliente final que usa la plataforma. Sus contribuciones principales se resumen así:

- **Reducción de errores fiscales:** Al garantizar la validez de RUC/DNI y demás datos antes de emitir comprobantes, se **evitan rechazos** por parte de la SUNAT u otros organismos. Esto reduce costos y demoras asociadas a corregir y reemitir facturas, mejorando el cumplimiento fiscal de la empresa usuaria.
- **Aumento de la productividad:** Funcionalidades como formularios de registro completos (que integran validación automática y autocompletado vía APIs) e importación masiva desde Excel **aceleran la gestión de clientes**. El personal invierte menos tiempo en limpiar datos o en cargas manuales, pudiendo enfocarse en tareas de valor (ventas, cobranzas).
- **Mejora de la trazabilidad y auditoría:** Tener un historial centralizado de todas las interacciones con un cliente (datos históricos, eventos de facturación, pagos, etc.) **facilita las auditorías internas y externas**, y ayuda a detectar fraudes o inconsistencias rápidamente. Por ejemplo, se puede auditar quién cambió la dirección de un cliente y cuándo, o listar todas las facturas anuladas de un cliente con sus motivos.

- **Potenciación del análisis de negocio:** La disponibilidad de **datos de clientes limpios y estructurados** (segmentados por tipo, con indicadores de actividad) permite realizar **segmentaciones avanzadas y definir estrategias de marketing** más efectivas. Un negocio puede identificar cuántos clientes mayoristas activos tiene, cuáles están inactivos (posibles churn), qué segmento genera más ingresos, etc., apoyando la toma de decisiones informada.
- **Facilidad de escalabilidad y mantenimiento:** Desde una perspectiva de arquitectura, el tener GestiónClientesBC como contexto independiente y bien definido significa que puede **evolucionar sin afectar a otros módulos**. Por ejemplo, si se debe integrar un nuevo servicio de verificación de identidad, se hace dentro de este contexto sin tocar la lógica de facturación. Esto aporta agilidad para adaptarse a cambios futuros (nuevas regulaciones de datos, nuevas funcionalidades solicitadas por usuarios, etc.) sin comprometer la estabilidad del sistema completo.

GestiónClientesBC no solo resuelve un problema operativo (gestionar clientes), sino que **aumenta la confiabilidad** del sistema de facturación, **mejora la satisfacción** tanto de usuarios (por menos errores) como de clientes finales (por atención más rápida y personalizada), y **sienta bases sólidas** para el crecimiento de la plataforma Factura Fácil y de los negocios que la utilizan.

# Aggregates, Entities, VO, Domains Events, Policies & Specifications\_gc

En esta sección se profundiza en **cada elemento del modelo de dominio** de GestiónClientesBC, detallando sus atributos, invariantes, comportamientos, así como los servicios de dominio y eventos que conectan la lógica de clientes con otros contextos. Además, se discuten algunas políticas específicas de negocio y especificaciones de validación.

## 2.1 Detalle de Elementos del Modelo de Dominio

### 2.1.1 Agregados (Aggregates)

**Agregado Cliente:** El único agregado identificado en este BC es *Cliente*, que funciona como **agregado raíz** y agrupa las entidades y VOs relacionados con un cliente. Atributos principales del agregado Cliente incluyen:

- **Identidad del Cliente:** usualmente un `clienteId` (UUID) interno para referencias técnicas, y el `DocumentoIdentidad` (tipo + número, VO) como identidad de negocio única del cliente.
- **Datos maestros:** `nombreCompleto` (para personas) o `razonSocial` (para empresas), dirección principal, correo principal, teléfono principal, etc. Son campos que describen al cliente. Algunos pueden ser opcionales dependiendo del tipo (por ejemplo, para empresa no hay nombre completo sino razón social).
- **Clasificación:** `tipoCliente` (VO enum) indicando categoría comercial, `monedaPredeterminada` y `formaPagoPredeterminada` (posiblemente como códigos a catálogos globales) para facilitar operaciones por defecto.
- **Estado:** `estadoCliente` (VO enum) que indica si está ACTIVO o INACTIVO (o ELIMINADO, según implementación).
- **Listas internas:** colecciones para `contactos` (lista de `ContactoCliente`), `adjuntos` (lista de `AdjuntoCliente`) y `operaciones` (lista de `OperacionCliente`) asociados.
- **Metadatos:** `fechaRegistro` (VO) indicando fecha de creación o última modificación del cliente; tal vez `usuarioCreacion` o `usuarioModificacion` para auditoría (no mencionado explícitamente, pero común en diseños auditables).

**Invariantes y Lifecycle:** El ciclo de vida del agregado Cliente inicia con la creación (*Alta de cliente*) donde se establecen datos obligatorios y se valida la identidad. Durante su vida, puede transicionar de ACTIVO a INACTIVO (**Deshabilitación**) y eventualmente a un estado eliminado (soft-delete). Sus invariantes clave son:

- `DocumentoIdentidad` permanece único e inmutable una vez asignado (cambiarlo implicaría crear otro cliente o generar un evento específico).
- Un cliente ACTIVO puede volverse INACTIVO, pero no viceversa sin una acción explícita de reactivación (no cubierta en requerimientos actuales).
- Si el cliente está INACTIVO, no se deben agregar nuevas `OperacionCliente` (salvo quizás anotaciones administrativas). Esto se refuerza externamente evitando facturararlo, etc.
- Si el cliente está marcado como eliminado, ya no debería aparecer en búsquedas normales ni aceptar modificaciones.

**Comportamientos (métodos de negocio) del agregado Cliente:**

- `crearCliente(datos)` – Método de fábrica/constructor que aplica al alta manual. Valida restricciones (RN-GC-001, RN-GC-002, etc.), setea estado ACTIVO por defecto, inicializa colecciones vacías y registra

FechaRegistro.

- `editarDatos(nuevosDatos)` – Aplica cambios a los campos editables. Antes de persistir, valida formateos (emails, DocumentoIdentidad si cambió, etc.) y unicidad si corresponde. Actualiza FechaRegistro y almacena los valores previos en una entidad de historial de cambios (si existiera) o prepara datos para evento ClienteModificado.
- `asignarTipo(tipoNuevo)` – Actualiza el VO TipoCliente del agregado. Verifica que el tipoNuevo esté dentro de los valores permitidos (es decir, coincide con uno de la enum TipoCliente). Después del cambio, podría afectar reglas de precios (por eso se emite ClienteModificado consumido por ListaPreciosBC).
- `deshabilitar(motivo)` – Cambia estadoCliente a INACTIVO. Antes comprueba que el cliente esté actualmente ACTIVO y que no tenga operaciones financieras pendientes (RN-GC-006). Registra la fecha de cambio (FechaRegistro nueva) y opcionalmente el motivo de deshabilitación para el evento.
- `eliminar()` – Realiza un soft-delete. Puede implementarse marcando un flag `eliminado=true` o moviendo el estado a ELIMINADO. Verifica invariantes: cliente puede estar ACTIVO o INACTIVO pero *debe* no tener operaciones pendientes ni adjuntos importantes. Tras marcar como eliminado, ya no se permiten modificaciones. Genera evento ClienteEliminado.
- `agregarContacto(contacto)` – Añade un nuevo ContactoCliente a la colección interna. Valida que el tipo de contacto no esté repetido si así se define (por ejemplo, podría permitir múltiples EMAIL\_SECUNDARIO o solo uno; en reglas no especificado), y valida formato según el tipo (ej. RN-GC-002 para email). Si ok, añade a la lista y genera evento ContactoAgregado.
- `editarContacto(contactoId, nuevoValor)` – Busca el contacto en la colección, valida existencia. Aplica cambio de valor (ej. actualizar el email secundario), valida formato. Genera evento ClienteModificado (porque se considera una modificación del cliente).
- `eliminarContacto(contactoId)` – Similar, remueve el contacto de la lista. Podría marcarlo como eliminado en vez de borrado físico. Genera evento ClienteModificado.
- `agregarAdjunto(archivo)` – Añade un AdjuntoCliente. Valida tipo y tamaño (ver restricción de adjuntos). Registra la fecha de subida. Genera evento AdjuntoAgregado.
- `eliminarAdjunto(adjuntoId)` – Remueve (o marca eliminado) un adjunto existente. Verifica que pertenezca al cliente y exista. Genera evento ClienteModificado (ya que es un cambio en la ficha).
- `registrarOperacion(datosOp)` – Crea una nueva OperacionCliente y la asocia. Internamente valida que el cliente no esté eliminado; si estuviera INACTIVO igual se registra la operación (pues puede ser una nota de crédito, etc.), pero podría haber decisiones de negocio sobre eso. Verifica RN-GC-005 (fecha). Después de agregar a la lista de operaciones, publica evento OperacionClienteRegistrada.

Como se ve, el agregado Cliente concentra la lógica y **protege sus invariantes**. Otras entidades (ContactoCliente, AdjuntoCliente, OperacionCliente) no exponen métodos de dominio hacia fuera; se modifican a través de los métodos del agregado, manteniendo consistencia. Esto implementa el patrón *Encapsulated Collections* y evita que desde fuera se manipulen contactos o adjuntos sin pasar por Cliente.

## 2.1.2 Entidades

Dentro del agregado Cliente tenemos varias **Entidades internas** (con identidad propia dentro del agregado). Ya las describimos conceptualmente; aquí las detallamos con sus atributos y relaciones:

- **ContactoCliente:** Entidad identificada por un `contactoId` único dentro del cliente. Atributos:
  - `tipoContacto` – (enum o VO) indica si es un correo secundario, teléfono, dirección u otro medio.
  - `valorContacto` – el dato en sí: email, número telefónico, texto de dirección, etc..

- **etiqueta** (opcional) – podría existir para identificar, por ejemplo "Oficina", "Casa" para direcciones, o "Personal" vs "Trabajo" en correos; no mencionado en doc pero a veces útil.
- Posiblemente **principal** (boolean) – en caso de permitir marcar uno de los contactos secundarios como principal en su categoría.

**Relaciones:** ContactoCliente pertenece a un Cliente (tiene una referencia a `clienteId` o, en implementación OO, es contenido en la colección del Cliente). No existe fuera del contexto del cliente.

**Reglas:** Formato del valor según tipo:

- Si `tipoContacto` = `EMAIL_SECUNDARIO`, aplicar val. RFC-5322 (RN-GC-002).
- Si `tipoContacto` = `TELEFONO_SECUNDARIO`, puede haber reglas de longitud o prefijos (no detalladas).
- Si `tipoContacto` = `DIRECCION`, se puede permitir texto libre, quizá con límite de longitud.

**Comportamientos:** La entidad en sí puede tener métodos para formatear su salida (e.g. concatenar "Tipo: valor"), pero las operaciones de agregar/editar/quitar están en Cliente. Al editar un ContactoCliente, se cambia su `valorContacto`. Si se quisiese cambiar el tipo, probablemente se eliminaría y agregaría uno nuevo.

- **AdjuntoCliente:** Entidad identificada por `adjuntoId`. Atributos típicos:

- `tipoAdjunto` – categoría del adjunto (ej: *CONTRATO, FOTO, DOCUMENTO\_IDENTIDAD, OTRO*).
- `nombreArchivo` – nombre original o descriptivo del archivo (ej. "Contrato\_ABC.pdf").
- `ruta` – path o URL donde está almacenado el archivo.
- `comentario` – texto opcional descriptivo (ej. "Contrato firmado en 2023").
- `fechaSubida` – timestamp de cuándo se adjuntó.
- `usuarioSubida` – posiblemente, para auditoría (no en doc, pero podría existir).

**Relaciones:** Pertenece a un Cliente; no tiene referencia a otras entidades.

**Reglas:** Ya mencionadas: el tipo debe ser uno de los permitidos, y usualmente el sistema validará que la extensión del archivo corresponda (e.g. si tipo=PDF, nombreArchivo termine en ".pdf"). Tamaño ya validado antes de crear la entidad. Si un cliente se elimina lógicamente, sus adjuntos podrían mantenerse para registro pero marcados, o eliminarse físicamente según políticas.

**Comportamientos:** Similar a Contacto, la agregación/retirada la maneja Cliente. La entidad podría tener un método para marcar como eliminado (`marcarEliminado()`), en caso de soft-delete de adjuntos, que quizás añada un flag en el objeto.

- **OperacionCliente:** Entidad identificada por `operacionId` (interno). Atributos principales:

- `tipoOperacion` – (VO `TipoOperacion`) tipo de operación histórica (factura, pago, etc.).
- `monto` – (VO `MontoOperacion`) importe de la operación.
- `fechaOperacion` – fecha/hora en que ocurrió la operación.
- `referenciaId` – (VO `ReferenciaId`) identificador del evento original en otro contexto (id de factura, id de pago).
- `detalle` (opcional) – pudiera almacenar información adicional como descripción de la operación o estado (por ej., si fue anulado, pendiente de pago, etc., aunque mucho de esto se deduce del tipo).

- **origen** (opcional) – quizás para indicar qué contexto generó la operación (Factura, Cobranza, etc., pero esto ya está en TipoOperacion implícitamente).

**Relaciones:** Pertenece a un Cliente. No suele referenciar a otras entidades dentro de GestiónClientesBC, pero conceptualmente se vincula a entidades de otros BC mediante referenciaId (por ej. una factura en ComprobantesElectonicosBC, un pago en ControlCajaBC).

**Reglas:** Una OperacionCliente es esencialmente inmutable: una vez registrada no se edita (salvo quizá agregar un campo de anulación si correspondiera anular una operación, pero en vez de eso se agrega otra operación de anulación). Regla RN-GC-005: fechaOperacion >= FechaRegistro cliente se chequea antes de guardar. Además, podría haber invariantes internas: e.g. un PAGO\_RECIBIDO debería tener monto <=0? (No, en pagos recibidos monto suele ser positivo como entrada de dinero, quizás negativo para devoluciones; esto no se menciona, la interpretación de signo probablemente esté estandarizada en cómo se calcula métricas: ej. Factura suma a cuentas por cobrar, Pago resta).

**Comportamientos:** Dado que se generan automáticamente, OperacionCliente tiene poco comportamiento expuesto. Podría tener métodos de ayuda para determinar si afecta saldo (e.g. `esCredito()` = true para pagos, false para facturas), o para formatear su presentación. La creación ocurre vía Cliente.registrarOperacion() que instancia la entidad.

En cuanto a **relaciones agregadas**, cabe reiterar: *ContactoCliente*, *AdjuntoCliente* y *OperacionCliente* son entidades anidadas en *Cliente*. **No tienen significado fuera del agregado** y por lo tanto su identidad completa podría ser compuesta (por ejemplo, *ContactoCliente* podría identificarse por una clave compuesta *clienteId+contactoId*). Esto garantiza que, por ejemplo, dos clientes distintos pueden tener *contactoId* = 1 en su propio ámbito sin colisión, ya que cada lista se maneja dentro de su cliente.

### 2.1.3 Value Objects (Definición e Invariantes)

Los **Value Objects** principales ya fueron enunciados en 1.5.2. Ampliaremos algunos detalles técnicos e invariantes específicos:

- **DocumentoIdentidad:** Compuesto de `tipoDocumento` + `numeroDocumento`. Su invariantes internas incluyen:
  - El `numeroDocumento` debe cumplir la longitud y formato según el `tipoDocumento`. Por ejemplo:
    - DNI: 8 dígitos numéricos exactos.
    - RUC: 11 dígitos numéricos; los dos primeros identifican tipo de contribuyente, el último es un dígito verificador calculado con un algoritmo estándar de SUNAT.
    - Pasaporte: puede incluir letras y números, longitud variable (ej. 2 letras + 6 dígitos, según país de emisión).
    - Carné de Extranjería: formato específico al país (generalmente alfanumérico).
    - etc.
  - Si aplica un dígito verificador (en RUC, ciertos carnés, etc.), el VO puede contener la lógica para calcularlo y verificarlo.
  - Es **inmutable**: si se quiere cambiar el documento de un cliente (caso raro, solo si se ingresó mal o la persona cambió de DNI a RUC, etc.), se haría creando otro VO y probablemente otro cliente (o con un caso de uso especial).

El método de comparación de DocumentoIdentidad considera iguales dos instancias si `tipoDocumento` y `numeroDocumento` son iguales (ignorando mayúsculas en caso de pasaporte, etc. si es alfanumérico). Esto permite usarlos como clave en colecciones, por ejemplo, para garantizar unicidad en la base de datos.

- **TipoDocumento:** Es un simple enum. Sus invariantes: solo puede tomar uno de los valores predefinidos (*DNI*, *RUC*, *CE*, *Pasaporte*, *Licencia*, etc.). Este VO se usa para condicionar lógicas; por ejemplo, `DocumentoIdentidad.validar()` puede ramificarse según TipoDocumento. No tiene atributos más allá de su nombre.
- **TipoCliente:** Enum igualmente fijo. Invariante: valor debe ser uno de los listados (*MAYORISTA*, *MINORISTA*, *DISTRIBUIDOR*, *REVENDEDOR*, ...). Podría estar abierto a extensión futura (e.g. agregar *VIP*, *Frecuente*, etc.), pero esas extensiones implicarían actualizar los consumidores (como *ListaPreciosBC*). TipoCliente se usa en reglas de precios y segmentación; por ende, es importante que cualquier cambio dispare un evento *ClienteModificado* para notificar a interesados.
- **EstadoCliente:** Enum con valores {*ACTIVO*, *INACTIVO*}. Podría tener un tercero *ELIMINADO* dependiendo de implementación (no se menciona explícitamente en doc, pero al hacer soft-delete probablemente hay que diferenciarlo). Invariante: Por defecto cuando se crea un cliente manual o automáticamente es *ACTIVO* (RN-GC-004 se asegura de eso en auto-creación). Solo operaciones autorizadas pueden cambiar este valor. Consumidores (como Facturación) deben respetar que *INACTIVO* significa no operable.
- **FechaRegistro:** Puede ser un wrapper de un `DateTime`. Invariante: debería actualizarse *solo* mediante lógicas de dominio (ej. no permitir setear manualmente, sino que se asigna la fecha del sistema en creación/modificación). Se expresa en un formato estandarizado (timestamp Unix, ISO string etc.). Ayuda a RN-GC-005 y para orden de operaciones.
- **TipoOperacion:** Enum con valores fijos. Invariante: Debe ser uno de los conocidos (*FACTURA\_EMITIDA*, *COTIZACION\_EMITIDA*, *PAGO\_RECIBIDO*, *FACTURA\_ANULADA*, etc.). Este VO se mantiene sincronizado con los tipos de eventos inbound que consume *GestionClientesBC*. Por ejemplo, si en un futuro se maneja un evento *NotaDeCreditoEmitida*, tendría que agregarse a *TipoOperacion*. A cada valor se le asocia posiblemente una lógica implícita:
  - *FACTURA\_EMITIDA* incrementa saldo por cobrar.
  - *PAGO\_RECIBIDO* decrementa saldo (o liquida facturas).
  - *FACTURA\_ANULADA* revierte una factura emitida (podría registrarse como operación separada o marcando la original).

Estas interpretaciones podrían hacerse en *IndicadoresNegocioBC*, pero el *TipoOperacion* provee la clasificación base.

- **MontoOperacion:** Proviene quizá de una clase genérica de *Money* con precisión fija. Invariante: Debe ser un número válido (no NaN ni infinito obviamente), con escala acorde a la moneda (por lo general 2 decimales para Soles o USD). Se puede implementar con *BigDecimal* para evitar errores de coma flotante. Podría contener la moneda también (aunque en doc parece que la moneda viene junto al evento *ComprobanteEmitido* más que por operación individual).
- Otra posible regla: en *Operaciones* de tipo pago, un monto negativo podría indicar devolución, pero generalmente se representaría como otro tipo de operación (ej. *PagoDevuelto*). Así que se espera monto  $\geq 0$  en la mayoría de casos (no explicitado, pero es razonable).
- **ReferenciaId:** VO (usualmente un string o UUID) sin mucha lógica más allá de su valor. Invariante: Podría ser nulo en casos donde no hay referencia (pero en la mayoría de *OperacionCliente* debería haberlo). En implementación, es importante para correlacionar y no se modifica después de asignado.

En conjunto, estos Value Objects permiten encapsular validaciones como:

- `DocumentoIdentidad` puede lanzar excepciones `DocumentoInvalidoException` si no cumple formato.
- `Email` (podría ser un VO separado, aunque en doc se maneja como string con validación).
- `Monto` podría asegurar su precisión decimal.

**Ejemplo de uso de VO (con JSON):** Si un usuario intenta crear un cliente con RUC incorrecto, el VO `DocumentoIdentidad` impedirá la creación. Supongamos un JSON de entrada:

```
1 {
2 "tipoDocumento": "RUC",
3 "numeroDocumento": "1234567890",
4 "nombre": "ABC SAC"
5 }
```

Aquí el RUC tiene 10 dígitos en vez de 11. Al intentar construir el VO `DocumentoIdentidad`, la validación fallaría (longitud incorrecta) y se retornaría un error (posiblemente un mensaje JSON con código de error "DocumentoInvalido"). Solo cuando el JSON contenga un RUC válido, por ejemplo:

```
1 {
2 "tipoDocumento": "RUC",
3 "numeroDocumento": "20601234571",
4 "razonSocial": "ABC S.A.C."
5 }
```

el servicio permitirá continuar. Este ejemplo ilustra cómo los VO garantizan consistencia antes de que los datos entren al sistema.

#### 2.1.4 Servicios de Dominio ↗

**ValidacionIdentidad (Servicio de Dominio):** Ya mencionado en 1.5.3, este es el servicio de dominio principal. Sus **métodos y contratos** pueden incluir:

- `validarIdentidadPorDocumento(tipoDoc, numero)` – Consulta la fuente adecuada (SUNAT si tipoDoc es RUC, RENIEC si es DNI, otros servicios o no disponible para pasaporte/licencia). **Entrada:** tipoDoc (ej. "RUC"), número (string). **Salida:** un objeto con los datos encontrados (ej. nombre, dirección, estado en padrón) o un error si no existe.
- `completarDatosCliente(cliente)` – Toma un objeto Cliente (posiblemente parcialmente llenado) y lo enriquece con datos externos. Por ejemplo, si el cliente solo tenía RUC y razón social vacía, rellena la razón social con lo devuelto por SUNAT. Este método podría ser llamado durante *Crear Cliente* si la opción de validación externa está habilitada.
- `verificarDocumentoIdentidad(documentoVO)` – Un método que encapsule la lógica de validación de dígito verificador para ciertos documentos (podría estar dentro del VO también).

**Contrato con APIs externas:** Este servicio es la capa intermedia entre el dominio y las APIs de terceros:

- Para RUC, consume la **API de SUNAT** (por ejemplo, un servicio REST público o mediante algún gateway empresarial). Envía un request con RUC y recibe JSON con razón social, dirección legal, estado (activo, habido) etc.
- Para DNI, consume la **API de RENIEC** (usualmente retorna nombre completo, apellido, etc.).

Este servicio de dominio debe interpretar esas respuestas y traducirlas al lenguaje ubicuo interno. Por ejemplo, la API SUNAT devuelve quizás `razonSocial` y `direccionFiscal` que mapea 1:1 con campos de Cliente; la API RENIEC devuelve `nombres` y `apellidoPaterno/Materno` que habría que concatenar para `nombreCompleto`.

#### Ejemplo de método y contrato (pseudocódigo):

```
1 public DatosIdentidad validarIdentidadPorDocumento(String tipoDoc, String numero)
2     throws DocumentoNoEncontradoException {
3     switch (tipoDoc) {
4         case "RUC":
5             return validarRUC(numero);
6         case "DNI":
7             return validarDNI(numero);
8         default:
9             // otros tipos no soportados por validación externa
10            return DatosIdentidad.vacio();
11    }
12 }
13
14 private DatosIdentidad validarRUC(String numero) throws DocumentoNoEncontradoException {
15     ResponseSunat res = sunatClient.getContribuyente(numero);
16     if (res == null) {
17         throw new DocumentoNoEncontradoException(numero);
18     }
19     return new DatosIdentidad(res.razonSocial, res.direccionFiscal);
20 }
21
22 private DatosIdentidad validarDNI(String numero) throws DocumentoNoEncontradoException {
23     ResponseReniec res = reniecClient.getPersona(numero);
24     if (res == null) {
25         throw new DocumentoNoEncontradoException(numero);
26     }
27     String nombreCompleto = String.join(" ", 
28         res.nombres,
29         res.apellidoPaterno,
30         res.apellidoMaterno
31     );
32     return new DatosIdentidad(nombreCompleto, res.direccion);
33 }
34 }
```

Donde `DatosIdentidad` es quizás un objeto simple con los campos que nos interesan completar.

**Coordinación de lógica compleja:** `ValidacionIdentidad` también podría implementar `reintentos` o decisiones: por ejemplo, si la API externa no responde, podría decidir permitir la creación del cliente con datos manuales pero dejando un flag de "pendiente de validar". Esa es una regla de negocio implícita: "*si API externa falla, continuar con datos ingresados*", que un servicio de dominio puede manejar (registrando el incidente en un log, etc.).

#### Otros Servicios de Dominio potenciales:

- **ServicioImportacionClientes:** Dado que la importación masiva involucra lógica (leer filas, validar cada una, crear o actualizar clientes existentes, acumular resultados), se podría implementar como servicio de dominio que use los repositorios de clientes para realizar estas operaciones en lote. Por ahora, la documentación sugiere que la importación se maneja como un caso de uso con integración a `XlsImportService`, pero conceptualmente podría haber un método `procesarArchivoClientes(file)` que devuelva un resumen de importación.

- **NotificacionesClientes:** No mencionado, pero un servicio de dominio podría encargarse de enviar notificaciones al área comercial cuando ciertos eventos ocurren (por ejemplo, notificar cuando se crea un cliente VIP, etc.). Esto típicamente se haría en IndicadoresNegocioBC o fuera del contexto, así que no profundizamos.

En síntesis, los servicios de dominio en GestiónClientesBC **mantienen la lógica de negocio pura que no cabe dentro de las entidades**, asegurando que el modelo permanezca cohesivo. ValidacionIdentidad es un claro ejemplo, integrándose con *Identity & Auth Domain* externamente para enriquecer el contexto de clientes.

### 2.1.5 Domain Events (Eventos de Dominio)

GestiónClientesBC publica varios **Eventos de Dominio** para notificar a otros contextos sobre hechos ocurridos en el dominio de clientes. Ya en la sección 1.6 se mencionaron algunos consumidores de estos eventos. Aquí listamos los eventos clave, su contenido y quién los produce/consume:

- **ClienteCreado:** Indica que se ha creado un nuevo cliente en el sistema.
  - *Productor:* GestiónClientesBC (agregado Cliente) al finalizar exitosamente el caso de uso *Crear Cliente* (o *Auto-crear Cliente*).
  - *Payload:* `clienteId` (identificador único), datos principales del cliente como `DocumentoIdentidad`, nombre/razón social, correo, teléfono, tipoCliente, fechaCreacion.
  - *Consumidores:* IndicadoresNegocioBC lo consume para agregar el nuevo cliente a sus estadísticas. La UI o portal puede consumirlo para actualizar listados en tiempo real. Posibles otros: sistemas externos interesados en nuevos clientes (integraciones CRM, etc.).
  - *Invariante:* Se emite solo una vez por cliente (no se repite).
- **ClienteModificado:** Notifica que uno o más datos de un cliente existente han sido cambiados.
  - *Productor:* GestiónClientesBC cuando se ejecuta *Editar Cliente*, *Asignar Tipo de Cliente*, *Gestionar Contacto* (editar/eliminar), *Gestionar Adjuntos* (eliminar), o *Validar Identidad Externa* y hay cambios persistentes. Básicamente cualquier cambio relevante excepto deshabilitar (que tiene evento propio) o eliminar (que opcionalmente tiene otro evento).
  - *Payload:* `clienteId`, y típicamente una estructura `camposModificados` listando qué atributos cambiaron con su valor anterior y nuevo, más `fechaCambio`. Por ejemplo: `camposModificados = [{campo: "correo", anterior: "old@example.com", nuevo: "new@example.com"}, {...}]`.
  - *Consumidores:* IndicadoresNegocioBC actualiza sus datos (por ej., si cambió TipoCliente, recalcula segmentación). La interfaz de usuario podría refrescar la ficha abierta del cliente si estaba viéndola. ListaPreciosBC consume específicamente cuando cambia TipoCliente para ajustar precios.
  - *Observaciones:* Este evento sirve como "catch-all" para cambios. En algunos sistemas se le llama *ClienteActualizado*; de hecho en el documento de métricas aparece así, pero conceptualmente es lo mismo.
- **ClienteDeshabilitado:** Señala que un cliente fue marcado como *Inactivo*.
  - *Productor:* GestiónClientesBC al completarse el caso de uso *Deshabilitar Cliente*.
  - *Payload:* `clienteId`, `motivoDeshabilitacion` (si se proporcionó alguno) y `fechaCambio`.
  - *Consumidores:* IndicadoresNegocioBC para excluirlo de sus análisis de clientes activos. ComprobantesElectrónicosBC para ya no permitir emitir nuevas facturas a este cliente. Otros sistemas que necesiten saber que el cliente ya no está disponible comercialmente (ej. un e-commerce integrado podría ocultarlo).

- *Observación:* A partir de este evento, el cliente queda bloqueado para operaciones nuevas. Cualquier evento *ClienteDeshabilitado* debería ser considerado por contextos que cachean info de clientes.
- **ClienteEliminado:** (No listado entre los 6 principales eventos en la doc, pero incluido en flujos). Indica que un cliente se eliminó lógicamente.
  - *Productor:* Gestión Clientes BC al ejecutar *Eliminar Cliente*.
  - *Payload:* `clienteId`, quizás `fechaEliminacion` y un indicador de quién realizó la acción.
  - *Consumidores:* Posiblemente Indicadores Negocio BC (para ya no considerar a ese cliente en absoluto) o para registros regulatorios. Dado que la deshabilitación ya había avisado, este evento suele ser interno. En la doc no se listó entre los eventos outbound principales (quizás por considerarse equivalente a Deshabilitado a efectos de otros contextos).
  - *Nota:* Podría no publicarse externamente en ciertos diseños, manteniéndose solo como registro interno.
- **ContactoAgregado:** Indica que se añadió un nuevo contacto secundario a un cliente.
  - *Productor:* Gestión Clientes BC en el caso de uso *Gestionar Contacto* (cuando es flujo de agregar).
  - *Payload:* `clienteId`, `contactoId` (id del nuevo contacto), `tipoContacto` y `valorContacto` añadido.
  - *Consumidores:* Principalmente la interfaz de usuario, para notificar al usuario que el contacto se agregó correctamente y tal vez reflejarlo en otras vistas. Podría haber un consumidor en Indicadores Negocio BC si se quiere medir cuántos clientes tienen email vs no, etc., pero es menos probable.
  - *Observaciones:* Este evento mejora la *experiencia de usuario* (feed en tiempo real) pero típicamente no afecta a otros contextos de negocio fuertemente (ya que contactos no son usados en otros BC, salvo quizás para envíos de correo en un contexto de notificaciones).
- **AdjuntoAgregado:** Indica que se subió un nuevo archivo adjunto al perfil de cliente.
  - *Productor:* Gestión Clientes BC en el caso de uso *Gestionar Adjuntos* (acción agregar).
  - *Payload:* `clienteId`, `adjuntoId`, `tipoAdjunto`, `nombreArchivo`, `ruta` (o URL) y `fechaSubida`.
  - *Consumidores:* Alguno de **servicio de almacenamiento/documentos** podría escucharlo para hacer copias de seguridad o auditar acceso. La UI podría ofrecer la descarga del archivo en cuanto recibe este evento. Es poco probable que otros contextos de negocio lo consuman, ya que es muy específico.
  - *Observaciones:* Garantiza respaldo documental: gracias a este evento, por ejemplo, un sistema de auditoría podría marcar que cierto contrato fue adjuntado a tal fecha.
- **OperacionClienteRegistrada:** Indica que se ha registrado una nueva operación en el historial del cliente.
  - *Productor:* Gestión Clientes BC al procesar un evento inbound de otro BC (factura emitida, pago recibido, etc.), es decir en el caso de uso *Registrar Operación en Historial*.
  - *Payload:* `clienteId`, `operacionId`, `tipoOperacion`, `montoOperacion`, `referenciaId` de la operación original, `fechaOperacion`.
  - *Consumidores:* Indicadores Negocio BC actualiza dashboards financieros (ventas totales, cobranzas al día, etc.) con esta nueva operación. Sistemas contables podrían también suscribirse para conciliar cuentas por cobrar. Módulos internos de auditoría registran este evento para trazabilidad.
  - *Observaciones:* Este evento es **crítico para mantener sincronizados los saldos** y reportes financieros entre contextos. Debe procesarse con garantía de entrega (para no perder registro de operaciones).

En total, los eventos **outbound** principales de GestiónClientesBC son 6: *ClienteCreado*, *ClienteModificado*, *ClienteDeshabilitado*, *ContactoAgregado*, *AdjuntoAgregado* y *OperaciónClienteRegistrada*. (**ClienteEliminado** sería un séptimo, menor uso). Estos eventos son inmutables, se publican inmediatamente tras completar el caso de uso correspondiente, y permiten una integración desacoplada con otros bounded contexts. En la siguiente sección (4) se documentan con más detalle incluyendo ejemplos JSON de cada evento.

## 2.2 Políticas de Negocio y Especificaciones (con ejemplos JSON)

Además de las reglas enumeradas en 1.7, existen **políticas de negocio** específicas que GestiónClientesBC implementa para validar datos y formatear información. Muchas de estas pueden conceptualizarse como *Especificaciones* en el sentido DDD (objetos o funciones booleanas que verifican criterios) o simplemente como validaciones encadenadas en los servicios de dominio. A continuación, se describen algunas políticas notables, ilustradas con ejemplos:

- **Política de Validación de Documento de Identidad:** Antes de crear o actualizar un cliente, se debe verificar que el documento proporcionado es válido. Esto se implementa combinando:
  - Una *especificación de formato* por tipo (parte de DocumentoIdentidad VO) – e.g. "Si tipo = RUC, debe tener 11 dígitos y pasar el algoritmo X".
  - Una *verificación de unicidad* – se consulta el repositorio para asegurar que ningún otro cliente activo tiene el mismo documento (esto puede hacerse mediante un índice único en base de datos o lógicamente antes de persistir).
- **Ejemplo JSON:** solicitud de creación:

```
1 {
2 "tipoDocumento": "DNI",
3 "numeroDocumento": "1234567",
4 "nombre": "Juan Perez"
5 }
```

Respuesta (error):

```
1 {
2 "error": "DocumentoInvalido",
3 "mensaje": "El DNI debe tener 8 dígitos."
4 }
```

Aquí la especificación detectó 7 dígitos en lugar de 8. Solo una vez que se corrija a "numeroDocumento": "12345678" la petición pasará ese check. Adicionalmente, si ese número ya existiera:

```
1 {
2 "error": "DocumentoDuplicado",
3 "mensaje": "Ya existe un cliente con DNI 12345678."
4 }
```

- **Política de Formato de Contacto:** Para contactos de tipo email y teléfono se aplican patrones de validación. Un spec podría encapsular la regex de email conforme RFC-5322. Para teléfonos, tal vez longitud mínima y caracteres numéricos.
  - **Ejemplo JSON:** agregar contacto:

```
1 {
2 "clienteId": "C100",
3 "tipoContacto": "EMAIL_SECUNDARIO",
4 "valorContacto": "juan_at_email.com"
5 }
```

Respuesta:

```
1 {
2 "error": "ContactoInvalido",
3 "mensaje": "Formato de email no válido."
4 }
```

Hasta que `valorContacto` sea algo como `"juan@mail.com"`, no se permitirá agregar.

- **Política de Tamaño de Archivo:** Antes de adjuntar un archivo, el adaptador de carga verifica el tamaño. Por ejemplo, si un archivo de 10MB se intenta subir:

- Respuesta de error (posible):

```
1 {
2 "error": "AdjuntoInvalido",
3 "mensaje": "El archivo excede el tamaño máximo de 5MB."
4 }
```

Esta verificación puede hacerse en la capa de interfaz (frontend) y también en el backend por seguridad.

- **Política de Tasas Fiscales (no directamente en GestiónClientesBC):** Cabe aclarar que aunque el nombre de la sección menciona tasas fiscales, en este BC específico no se manejan impuestos (eso es más de ComprobantesElectrónicosBC). Sin embargo, la validez de RUC sí está relacionada a situación fiscal: un RUC puede estar "activo/habido" o no en SUNAT. Una posible mejora futura de las políticas sería:
  - No solo validar formato de RUC, sino también consultar a SUNAT si ese RUC está activo en el padrón de contribuyentes. De estar "suspendido" o "dado de baja", quizás alertar al usuario o impedir ciertas operaciones. Esto entra en políticas de negocio de mayor nivel, integrando la validación externa como criterio de calidad de cliente.
- **Política de Auditoría de Cambios:** Cada vez que se modifica un cliente, se registra internamente la diferencia antes vs después (como vimos con camposModificados en ClienteModificado event). Esto no es visible para otros contextos excepto Indicadores, pero es parte de la especificación interna: *"todo cambio en datos maestros genera un registro de auditoría"*. En JSON interno podría verse algo así:

```
1 {
2 "clienteId": "C100",
3 "cambio": { "campo": "telefono", "antes": "987654321", "despues": "999888777" },
4 "fecha": "2025-07-04T17:00:00Z"
5 }
```

Este no es un evento público sino un registro que podría guardarse en una colección de historial de versiones.

En general, Gestión de Clientes BC se asegura de que **ningún dato crítico ingrese sin validar** y que las políticas definidas (formato documento, unicidad, formatos de contacto, límites de tamaño, estados coherentes) se apliquen consistentemente en todos los casos de uso. Muchas de estas políticas se implementan en código dentro de métodos de entidades o servicios de dominio, y su cumplimiento es verificado mediante pruebas unitarias (ver sección 14, estrategia de testing).

En las secciones siguientes, veremos cómo estas reglas y elementos de dominio se orquestan en casos de uso específicos y cómo los eventos llevan información entre contextos.

## Casos de Uso\_gc

A continuación se describen **detalladamente los casos de uso** soportados por Gestión Clientes BC, incluyendo el flujo principal de cada uno, variantes, entradas, salidas y cómo interactúan con otros componentes. Para mayor claridad, se incluyen diagramas de secuencia que ilustran la interacción entre el **actor principal** (usuario u otro sistema) y Gestión Clientes BC, así como cualquier sistema externo involucrado (p. ej. APIs de SUNAT) u otros contextos (p. ej. Comprobantes Electrónicos BC enviando eventos).

**Lista de Casos de Uso en Gestión Clientes BC:** (extraída del documento de requisitos)

1. **Crear Cliente** – Registro manual de un nuevo cliente. Actor: Usuario del sistema.
2. **Auto-crear Cliente desde Comprobante** – Creación automática de cliente a partir de un evento de facturación. Actor: Subsistema de Facturación.
3. **Editar Cliente** – Modificación de datos de un cliente existente. Actor: Usuario.
4. **Asignar Tipo de Cliente** – Clasificación del cliente en una categoría. Actor: Usuario.
5. **Deshabilitar Cliente** – Marcar un cliente como inactivo. Actor: Usuario.
6. **Eliminar Cliente** – Eliminación lógica (soft-delete) de un cliente. Actor: Usuario.
7. **Gestionar Contacto** – Añadir/editar/eliminar un contacto secundario de cliente. Actor: Usuario.
8. **Gestionar Adjuntos** – Adjuntar o eliminar archivos relacionados al cliente. Actor: Usuario.
9. **Validar Identidad Externa** – Verificar datos de cliente mediante servicios externos. Actor: Usuario / Sistema interno.
10. **Registrar Operación en Historial** – Registrar automáticamente una operación (factura/pago) en el historial del cliente al recibir un evento. Actor: Subsistema de Eventos (otros BC).
11. **Consultar Historial de Operaciones** – Consultar y filtrar el historial de operaciones de un cliente. Actor: Usuario / Auditor.
12. **Importar Clientes desde CSV/Excel** – Cargar en lote múltiples clientes desde un archivo. Actor: Usuario.
13. **Consultar Cliente** – Obtener la ficha completa de un cliente (datos maestros + asociados). Actor: Usuario / Vendedor.

A continuación, cada caso de uso se presenta con su **descripción, precondiciones, flujo principal de eventos** paso a paso, **flujos alternativos** en caso de errores o condiciones especiales, **eventos de dominio generados, entidades involucradas**, y diagramas de secuencia ASCII/mermaid para ilustrar la interacción.

### 3.1 Detalle de cada Caso de Uso

#### 3.1.1 Crear Cliente

**Actor Principal:** Usuario del sistema (por ejemplo, un administrador o personal encargado de mantenimiento de clientes).

**Propósito:** Permitir registrar manualmente un nuevo cliente con todos sus datos en el sistema.

**Precondiciones:**

- El usuario debe estar autenticado y tener permiso de creación de clientes (rol con privilegio `GestionClientes.Crear`).

- El documento de identidad proporcionado para el nuevo cliente no debe existir ya registrado en el sistema (unicidad).
- Se cuenta con la información mínima requerida del cliente: nombre o razón social, tipo y número de documento, etc.

#### **Flujo Principal:**

1. El **Usuario** ingresa en la interfaz los datos del nuevo cliente: Tipo de documento, Número de documento, Nombre o Razón Social, Correo electrónico, Teléfono, Dirección postal, Tipo de cliente, Moneda predeterminada, Forma de pago predeterminada, etc..
2. El **Sistema (GestionClientesBC)** valida que se hayan llenado los campos obligatorios y que el formato del DocumentoIdentidad es correcto (regla RN-GC-001, RN-GC-002). Si algún dato requerido falta o tiene formato inválido, se cancela la operación (ver Flujos Alternativos).
3. [Opcional] Si está habilitada la verificación externa de identidad, el sistema invoca al **servicio externo SUNAT/RENIEC** para completar y validar los datos fiscales del cliente (por ejemplo, autocompletar razón social a partir de RUC). Esto ocurre en background sin bloquear la interfaz, usualmente.
4. El **Sistema** crea una nueva instancia del **agregado Cliente** en estado ACTIVO, asignando un nuevo `clienteId` (UUID). Se almacenan los datos ingresados, se calcula y setea el `FechaRegistro` actual automáticamente.
5. El **Sistema** persiste el nuevo cliente en el repositorio de clientes y **publica un evento de dominio ClienteCreado** para notificar a otros contextos que un cliente ha sido añadido.
6. El **Sistema** devuelve al actor un resultado indicando éxito – típicamente el `clienteId` recién creado y un mensaje de confirmación. La interfaz podría redirigir a la vista de detalle del cliente creado.

#### **Flujos Alternativos:**

- 2a. *Formato de documento inválido:* Si el número de RUC/DNI no cumple el patrón esperado (por ejemplo, letras en DNI o dígitos incorrectos en RUC), el sistema rechaza la creación, mostrando un error "Documento de Identidad inválido". (RN-GC-001).
- 2b. *Cliente duplicado:* Si ya existe un cliente con el mismo tipo y número de documento, se aborta la operación con error "Cliente duplicado". (RN-GC-001 sobre unicidad).
- 3a. *API externa falla:* Si la consulta a SUNAT/RENIEC no responde o encuentra inconsistencia, el sistema puede registrar el fallo en un log pero **continuar** con la creación usando solo los datos manuales ingresados. (El usuario podría ver una advertencia "No se pudo validar con SUNAT, verifique posteriormente").
- 5a. *Error en base de datos:* Si por alguna razón la inserción en la base de datos falla (p. ej. problema de conexión), el sistema revierte la operación y notifica al usuario. Este caso no está en el doc pero es un alternativo técnico usual.

**Eventos Generados:** `ClienteCreado` – contiene los datos clave del nuevo cliente (id, doc, etc.).

**Entidades Afectadas:** `Cliente` (nuevo agregado creado en repositorio).

#### **Diagrama de Secuencia:**



*Figura 1: Flujo del caso de uso "Crear Cliente". El usuario ingresa los datos, el sistema valida y (opcionalmente) consulta fuentes externas antes de crear el cliente y notificar a otros contextos.*

### 3.1.2 Auto-crear Cliente desde Comprobante

**Actor Principal:** Subsistema de Facturación (ComprobantesElectonicosBC), actuando a través de un evento automáticamente.

**Propósito:** Cuando en el proceso de facturación electrónica se emite un comprobante con un cliente no registrado, este caso de uso permite **crear un cliente básico en tiempo real**, para luego asociar la factura a ese nuevo cliente.

#### Precondiciones:

- Se recibe un evento `ComprobanteEmitido` del BC de ComprobantesElectonicos con un `clienteId` nulo o no reconocido (desconocido para GestiónClientesBC).
- El evento contiene al menos información mínima del cliente en el comprobante (por ejemplo, número de documento y tipo, y posiblemente nombre/dirección obtenidos en el comprobante).

#### Flujo Principal:

1. El **evento** `ComprobanteEmitido` es disparado por ComprobantesElectonicosBC al emitirse un comprobante de venta. Este evento ingresa a GestiónClientesBC asincrónicamente mediante su suscripción configurada.
2. El **Sistema (GestiónClientesBC)** recibe el evento y detecta que el `clienteId` incluido es desconocido (por convención, podría venir nulo o con un valor especial).
3. El **Sistema** procede a **crear un Cliente mínimo** con la información disponible en el evento:
  - Usa el tipo y número de documento proporcionados (por ejemplo, RUC del comprobante).
  - Asigna `EstadoCliente = ACTIVO` por defecto (RN-GC-004).
  - Otros campos como nombre/razón social se toman del comprobante si están, o se dejan vacíos o con un marcador (ej. "Cliente sin nombre").
  - Genera un nuevo `clienteId`.
4. El **Sistema** persiste el nuevo cliente en la base de datos de clientes.
5. El **Sistema publica el evento** `ClienteCreado` correspondiente a esta auto-creación, para que otros contextos (Indicadores, etc.) sepan que se añadió un cliente.
6. (*No hay respuesta a un actor humano en este flujo, pero internamente, ComprobantesElectonicosBC continuará procesando su factura usando el cliente recién creado.*)

#### Flujos Alternativos:

- 2a. *DocumentoIdentidad inválido:* Si el documento en el comprobante es claramente inválido (ej. RUC con dígitos de más o texto no numérico), GestiónClientesBC **podría descartar la auto-creación** y notificar al módulo de facturación que el cliente no pudo crearse. En la práctica, ComprobantesElectonicosBC tendría que manejar esa situación, tal vez marcando la factura para revisión manual.
- 3a. *Cliente ya existe:* Si por alguna condición de carrera, cuando llega el evento ya alguien registró ese documento, la auto-creación debería reconocerlo. Podría entonces simplemente mapear la factura al cliente existente y no crear duplicado. Este caso no se menciona en doc, pero es deseable prevenir duplicados en auto-creación.

**Eventos Generados:** `ClienteCreado` (igual que en caso manual).

**Entidades Afectadas:** `Cliente` (nuevo creado), posiblemente solo con `DocumentoIdentidad` y `estado`, sin detalles completos.

#### Diagrama de Secuencia:



Figura 2: Flujo "Auto-crear Cliente desde Comprobante". El subsistema de facturación emite un evento; GestiónClientesBC crea al vuelo un nuevo cliente con los datos mínimos disponibles.

### 3.1.3 Editar Cliente

**Actor Principal:** Usuario del sistema (por ejemplo, personal de soporte que necesita actualizar datos).

**Propósito:** Permitir la **actualización** de la información de un cliente existente, por ejemplo cambiar su dirección, email, teléfono, tipo de cliente, etc..

#### Precondiciones:

- El cliente a editar **existe** en el sistema y está en estado ACTIVO (no eliminado; puede estar inactivo pero generalmente solo se editan activos).
- El usuario tiene permisos de edición sobre clientes.
- Se conoce el `clienteId` del cliente que se desea modificar.

#### Flujo Principal:

1. El **Usuario** solicita editar un cliente específico, proporcionando el `clienteId` y los campos que desea modificar con sus nuevos valores (por ejemplo, nuevo correo, nueva dirección, o cambio de tipoCliente).
2. El **Sistema** (GestiónClientesBC) busca el cliente en su repositorio y verifica que exista y (opcionalmente) que su estado actual permita edición (idealmente ACTIVO).
3. El **Sistema** valida cada dato proporcionado:
  - Formato de campos como correo (RFC-5322) si fue cambiado.
  - Si se pretende cambiar el DocumentoIdentidad (caso poco común), validaría formato y unicidad también (RN-GC-001).
  - Cualquier otro campo con reglas (por ej. no poner caracteres inválidos en nombre, etc.).
4. El **Sistema** aplica los cambios al agregado Cliente: actualiza los campos, y registra internamente la FechaRegistro como momento de última modificación.
5. El **Sistema publica un evento** `ClienteModificado` con detalle de qué campos cambiaron. Por ejemplo, incluirá el correo anterior y nuevo si fue uno de los modificados.
6. El **Sistema** confirma al usuario la edición con éxito, devolviendo los datos actualizados del cliente o simplemente un mensaje de éxito.

#### Flujos Alternativos:

- 1a. *Usuario sin permisos:* Si el actor no tiene el rol adecuado, se lanza una excepción de autorización y no se permite la edición.
- 2a. *Cliente no existe:* Si el `clienteId` proporcionado no se encuentra, se devuelve error "Cliente no encontrado".
- 3a. *Email inválido:* Si el nuevo correo no pasa la validación RFC-5322, se rechaza la operación con `ContactoInvalidoException` o mensaje equivalente.
- 3b. *Intento de duplicar DocumentoIdentidad:* Si el usuario intenta cambiar el documento a uno que ya existe en otro cliente, error "Documento ya registrado".
- 3c. *Campos sin cambios:* Si ninguno de los valores difiere del existente, el sistema podría optar por no generar evento (o generar un `ClienteModificado` con `camposModificados` vacío). Podría notificar "No se realizaron cambios".

- 5a. *Notificar solo cambios relevantes*: Si algunos campos críticos se modificaron (ej. tipoCliente), puede haber lógica para acciones adicionales en otros BC (pero eso se maneja por evento ya).

**Eventos Generados:** ClienteModificado – contendrá el clientId y la lista de campos modificados con sus valores previos y nuevos.

**Entidades Afectadas:** Cliente (actualizado), posiblemente cascada a algún VO interno como DocumentoIdentidad si fue cambiado, etc. Entidades ContactoCliente, AdjuntoCliente no se tocan aquí (casos separados).

#### Diagrama de Secuencia:



Figura 3: Flujo "Editar Cliente". El usuario envía los cambios, el sistema valida y actualiza el registro, luego emite un evento de modificación.

#### 3.1.4 Asignar Tipo de Cliente

**Actor Principal:** Usuario del sistema (por ejemplo, un administrador de ventas que segmenta clientes).

**Propósito:** Cambiar la **categoría de negocio** de un cliente (su TipoCliente) para fines de segmentación y reglas de precios.

#### Precondiciones:

- El cliente existe (puede estar activo; incluso si estuviera inactivo, su tipo podría cambiar, pero generalmente se hace en activos).
- El nuevo TipoCliente a asignar es válido y diferente al actual del cliente.
- Usuario con permisos de modificar cliente.

#### Flujo Principal:

1. El **Usuario** selecciona un cliente (clientId) y elige un nuevo **TipoCliente** para asignarle (por ejemplo cambiar de MINORISTA a MAYORISTA).
2. El **Sistema** verifica que el cliente existe y carga su información actual.
3. El **Sistema** comprueba que el nuevo TipoCliente está dentro de los valores permitidos (es uno de la enumeración conocida).
4. El **Sistema** actualiza el campo TipoCliente del Cliente al valor nuevo.
5. El **Sistema publica un evento** ClienteModificado para reflejar este cambio de segmento. Alternativamente, podría haber un evento más específico (ClienteReclasificado), pero en doc se usa ClienteModificado.
6. El **Sistema** confirma al usuario que la clasificación se ha actualizado exitosamente.

#### Flujos Alternativos:

- 3a. *Tipo no reconocido*: Si el tipo seleccionado no coincide con ninguno de la lista (p. ej. error en UI o datos corruptos), se lanza una excepción de validación (ValueObjectValidationException) y no se aplica el cambio.
- 2a. *Cliente inexistente*: Manejo estándar, error "Cliente no encontrado".
- 4a. *Tipo igual al actual*: Si el TipoCliente nuevo es el mismo que ya tenía el cliente, es posible que el sistema no haga nada (podría retornar un mensaje "El cliente ya es de tipo X"). Este caso no se menciona, pero conviene considerarlo para evitar eventos innecesarios.

**Eventos Generados:** ClienteModificado (con detalle del campo TipoCliente anterior y nuevo).

**Entidades Afectadas:** Cliente (campo tipoCliente modificado).

## Diagrama de Secuencia:



1 mermaid

Figura 4: Flujo "Asignar Tipo de Cliente". Un usuario reclasifica a un cliente y el sistema actualiza su tipo y notifica a contextos interesados (e.g. ListaPreciosBC).

### 3.1.5 Deshabilitar Cliente

**Actor Principal:** Usuario del sistema (por ejemplo, para bloquear clientes morosos o inactivos).

**Propósito:** Cambiar el estado de un cliente a INACTIVO, evitando que se use en nuevas transacciones.

#### Precondiciones:

- El cliente debe existir y estar actualmente ACTIVO (ya que no tendría sentido deshabilitar uno ya inactivo).
- El cliente no debe tener operaciones pendientes de completar (p. ej. facturas por cobrar sin pagar), según política interna.
- Usuario autorizado para esta acción.

#### Flujo Principal:

1. El **Usuario** indica la intención de deshabilitar un cliente, proporcionando su `clienteId` y opcionalmente un `motivoDeshabilitacion` (razón por la cual se inactiva, ej. "Cliente dejó de operar").
2. El **Sistema** verifica que el cliente existe y que su `EstadoCliente` actual es ACTIVO.
3. El **Sistema** comprueba que no haya operaciones pendientes asociadas al cliente. Esto puede implicar:
  - Consultar el módulo de facturación o cobranzas para ver si tiene facturas impagadas, o
  - Revisar su historial de OperacionCliente por si hay señales de pagos pendientes, etc.De acuerdo al doc: "verifica no hay operaciones pendientes (RN-GC-006)". Si encuentra pendientes, aborta (flujo alterno).
4. El **Sistema** actualiza el estado del cliente a INACTIVO y registra la fecha/hora de deshabilitación (`FechaRegistro`). También almacena el motivo si se proporcionó.
5. El **Sistema publica un evento** `ClienteDeshabilitado` con el `clienteId`, motivo y fecha.
6. El **Sistema** confirma al usuario que el cliente fue deshabilitado exitosamente. En la interfaz el cliente podría aparecer marcado como inactivo.

#### Flujos Alternativos:

- 3a. *Operaciones pendientes:* Si el cliente tiene deudas u operaciones abiertas, se cancela la acción con un error tipo "Cliente con cuentas pendientes, no se puede deshabilitar". (Internamente podría lanzarse `ClienteConCuentasPendientesException` ).
- 2a. *Cliente ya inactivo:* Si de alguna forma se intenta deshabilitar un cliente no ACTIVO (posiblemente un error de uso), el sistema podría simplemente ignorar la solicitud o advertir "Cliente ya estaba inactivo".
- *General:* Errores de autenticación/autorización se manejan previamente (no llega al flujo principal si no tiene permiso).

**Eventos Generados:** `ClienteDeshabilitado`.

**Entidades Afectadas:** `Cliente` (cambio de estado). Podría considerarse también que a nivel de OperacionCliente, no se generan más para este cliente tras esto (pero no es una entidad modificada directamente en esta acción).

#### Diagrama de Secuencia:



Figura 5: Flujo "Deshabilitar Cliente". El sistema comprueba que el cliente esté libre de operaciones pendientes antes de inactivarlo y notificar la deshabilitación.

#### 3.1.6 Eliminar Cliente

**Actor Principal:** Usuario del sistema (administrador con privilegios altos, dado que elimina registros).

**Propósito:** **Eliminar lógicamente** un cliente del sistema, típicamente tras confirmar que no tiene deudas ni movimientos recientes, cumpliendo políticas de retención.

#### Precondiciones:

- El cliente existe y puede estar ACTIVO o INACTIVO (en doc dice "EstadoCliente = INACTIVO o ACTIVO" como permitido) – probablemente se sugiere deshabilitar primero, pero no es obligatorio.
- No debe tener operaciones/relaciones activas: no tener facturas o contratos vigentes, etc. (Similar a RN-GC-006).
- Usuario con permisos especiales para eliminar.

#### Flujo Principal:

1. El **Usuario** solicita la eliminación lógica de un cliente dado (`clientId`). Esta acción suele requerir confirmación explícita (fuera de este flujo).
2. El **Sistema** verifica que el cliente existe y su estado actual sea ACTIVO o INACTIVO (no eliminado ya).
3. El **Sistema** comprueba que **no existan registros activos asociados** que impidan eliminar: no tener operaciones en historial que deban conservarse, o si las hay, que se cumplieron las políticas de retención (por doc: "No hay operaciones activas"). También podría checar que no tenga adjuntos críticos.
4. El **Sistema** cambia el estado del cliente a un estado representando "Eliminado" (podría reutilizar INACTIVO con un flag, o un estado separado) y marca una propiedad de `eliminado = true`. Registra la acción en auditoría (usuario, fecha).
5. El **Sistema publica un evento** `ClienteEliminado` (soft-delete), aunque como notamos, este evento no estaba en la lista principal, podría o no emitirse dependiendo de decisiones de diseño.
6. El **Sistema** notifica al usuario que el cliente ha sido eliminado lógicamente. A partir de aquí, el cliente puede dejar de mostrarse en listados normales (solo consultable vía funciones admin/auditoría).

#### Flujos Alternativos:

- 3a. *Operaciones asociadas existen:* Si el cliente tiene historial con registros que por políticas se deben conservar, la eliminación se impide: "No se puede eliminar cliente con operaciones registradas". (Doc: RN-GC-006 también cubre esto, similar a deshabilitar).
- 2a. *Cliente ya eliminado:* Si se intenta eliminar un cliente ya eliminado previamente (no debería ocurrir vía UI, pero si ocurre), sistema puede ignorar o informar "Cliente ya eliminado".
- *General:* Si se viola alguna política de retención (ej., el cliente tiene facturas en los últimos 5 años y la política dice conservarlo), se aborta.

**Eventos Generados:** `ClienteEliminado` (soft-delete).

**Entidades Afectadas:** `Cliente` (estado cambiado, flag eliminado). Posiblemente, también se podrían marcar sus Contactos y Adjuntos como eliminados cascada, pero no es indispensable.

#### Diagrama de Secuencia:



Figura 6: Flujo "Eliminar Cliente". Un administrador elimina lógicamente un cliente tras pasar todas las validaciones; el cliente queda marcado como eliminado.

### 3.1.7 Gestionar Contacto

**Actor Principal:** Usuario del sistema (por ejemplo, personal de administración de clientes).

**Propósito:** Añadir, actualizar o eliminar información de contacto secundaria de un cliente (emails alternos, teléfonos adicionales, direcciones).

Este caso de uso abarca tres sub-flujos: **Agregar Contacto**, **Editar Contacto**, **Eliminar Contacto**.

#### Precondiciones (generales):

- El cliente en cuestión existe (no eliminado).
- Usuario con permiso para editar clientes (los contactos se consideran parte de la ficha de cliente).
- Para editar/eliminar: el contacto específico debe existir asociado al cliente.

#### Flujo Principal (Agregar Contacto):

1. El **Usuario** ingresa los datos de un nuevo contacto para un cliente dado: especifica el `clienteId`, el `tipoContacto` (EMAIL\_SECUNDARIO, TELEFONO\_SECUNDARIO o DIRECCION) y el `valorContacto` correspondiente (ej. la dirección de email o el número telefónico).
2. El **Sistema** verifica que el cliente existe y recupera su instancia (agregado Cliente).
3. El **Sistema** valida el formato del valor proporcionado según el tipo:
  - Si es EMAIL\_SECUNDARIO, aplica validación de email (RN-GC-002).
  - Si es TELEFONO\_SECUNDARIO, podría validar solo dígitos o longitud.
  - Si es DIRECCION, usualmente no hay un formato estricto salvo quizás longitud máxima.  
Esto corresponde al paso "Validate formato".
4. El **Sistema** crea un nuevo objeto ContactoCliente con tipo y valor dados, y lo agrega a la colección de contactos del Cliente.
5. El **Sistema publica un evento** `ContactoAgregado` con los detalles del nuevo contacto (`clienteId`, `contactoId`, `tipo`, `valor`).
6. El **Sistema** confirma al usuario que el contacto fue agregado exitosamente.

#### Flujo Principal (Editar Contacto):

1. El **Usuario** envía una solicitud para modificar un contacto existente: proporciona el `contactoId` (o identifica el contacto por tipo+valor actual) y el nuevo `valorContacto`.
2. El **Sistema** busca el contacto en la colección del Cliente (por `clienteId` y `contactoId`).
3. Valida el nuevo valor (formato igual que en agregar).
4. Actualiza el campo valor del ContactoCliente con el nuevo dato.
5. Genera un evento `ClienteModificado` (ya que un contacto cambiado se considera cambio en la ficha del cliente). *Nota:* Doc indica ClienteModificado para editar/eliminar contacto.
6. Confirma al usuario la edición.

#### Flujo Principal (Eliminar Contacto):

1. El **Usuario** solicita eliminar un contacto, indicando el `contactoId` (o seleccionándolo de la lista del cliente).

2. El **Sistema** localiza el contacto y verifica su existencia.
3. Elimina el contacto de la colección (o marca como eliminado).
4. Publica evento `ClienteModificado` (por cambio en la ficha).
5. Confirma al usuario que el contacto fue eliminado.

#### **Flujos Alternativos:**

- **Agregar/Edit:** Formato inválido → lanza `ContactoInvalidoException` y no se agrega/actualiza.
- **Editar/Eliminar:** Contacto no encontrado (quizás ya eliminado manualmente por otro) → lanza `EntityNotFoundException`.
- **Agregar duplicado:** Si la política no permite duplicar (ej. ya hay un EMAIL\_SECUNDARIO igual), podría rechazar con error "Contacto ya existe". (No mencionado, pero plausible).
- **Permisos:** Sin permisos de edición, rechaza la acción.

#### **Eventos Generados:**

- `ContactoAgregado` (cuando se añade).
- `ClienteModificado` (cuando se edita o elimina un contacto).

**Entidades Afectadas:** `ContactoCliente` (nuevo o modificado/eliminado) y su contenedor `Cliente` (se considera modificado).

#### **Diagrama de Secuencia (Agregar Contacto como ejemplo):**



Figura 7: Flujo "Gestionar Contacto" (subflujo agregar). El sistema valida y agrega un contacto, notificando con evento dedicado.

#### **3.1.8 Gestionar Adjuntos**

**Actor Principal:** Usuario del sistema.

**Propósito:** Permitir **adjuntar archivos** al perfil de un cliente (ej. subir un contrato firmado) o **eliminar** adjuntos existentes.

Sub-flujos: **Agregar Adjunto**, **Eliminar Adjunto**.

#### **Precondiciones:**

- Cliente existe.
- Para agregar: disponer del archivo (tipos permitidos PDF/JPG/PNG, tamaño ≤5MB).
- Para eliminar: adjunto existente identificado.

#### **Flujo Principal (Agregar Adjunto):**

1. El **Usuario** selecciona un archivo para adjuntar y proporciona metadatos: `clienteId`, `tipoAdjunto` (CONTRATO, FOTO, etc.), `nombreArchivo` (nombre original), `ruta` o el archivo en sí (que será almacenado), y un `comentario` opcional.
2. El **Sistema** valida el archivo: comprueba el **tipo de archivo** y su **tamaño**:
  - Asegura que la extensión o MIME corresponda a PDF/JPG/PNG.
  - Verifica que el tamaño sea <= 5MB (RN: restricción de adjuntos).
 Si alguna validación falla, rechaza (flujo alterno).

3. El **Sistema** guarda físicamente el archivo (o delega a un servicio de almacenamiento) y obtiene/almacena la ruta o URL de acceso.
4. Crea un objeto **AdjuntoCliente** con tipo, nombre, ruta y asocia la fecha actual como `fechaSubida`.
5. Agrega el AdjuntoCliente a la colección de adjuntos del Cliente.
6. Publica un evento **AdjuntoAgregado** con detalles del adjunto (clienteId, adjuntoId, tipo, nombreArchivo, ruta, fecha).
7. Notifica al usuario que el adjunto se subió exitosamente.

#### **Flujo Principal (Eliminar Adjunto):**

1. El **Usuario** solicita eliminar un adjunto, especificando el `adjuntoId` (o identificándolo de alguna forma).
2. El **Sistema** busca el adjunto en la lista del cliente y valida que existe y que pertenece a ese cliente.
3. El **Sistema** elimina lógicamente el adjunto:
  - Podría marcar un flag `activo=false` en AdjuntoCliente o removerlo.
  - Especifica esto como soft-delete (no necesariamente borra el archivo físico inmediatamente, por política de retención).
4. Publica un evento **ClienteModificado** para indicar cambio en la ficha del cliente (cantidad de adjuntos reducida). (*Doc mostraba erroneamente "ClienteEliminado con cambio de adjuntos" pero en la lista de eventos figuran ClienteModificado para Eliminar adjunto*).
5. Notifica al usuario que el adjunto fue eliminado.

#### **Flujos Alternativos:**

- 2a. *Formato o peso inválido (Agregar)*: Si el archivo no es PDF/JPG/PNG o excede tamaño, se rechaza con error "Adjunto inválido".
- 2b. *Adjunto no encontrado (Eliminar)*: Si el adjunto ya no existe (quizá fue removido por otro usuario o nunca existió ese ID), lanza `EntityNotFoundException`.
- *Almacenamiento falla*: Si no se puede guardar el archivo (espacio insuficiente, error I/O), retorna error y no crea AdjuntoCliente.
- *Permisos*: Solo ciertos roles pueden adjuntar/eliminar (podría haber restricción de que cajeros no adjunten, etc.), pero no mencionado.

#### **Eventos Generados:**

- **AdjuntoAgregado** al subir.
- **ClienteModificado** al eliminar.

**Entidades Afectadas:** `AdjuntoCliente` (nuevo o eliminado), y su agregado `Cliente` (modificado por cambio en colecciones).

#### **Diagrama de Secuencia (Agregar Adjunto):**



(El flujo de eliminación es similar a Contacto: se busca y quita, con evento `ClienteModificado`.)

#### **3.1.9 Validar Identidad Externa**

**Actor Principal:** Usuario del sistema o Sistema interno (puede ser invocado automáticamente tras crear cliente, por eso se menciona sistema interno).

**Propósito:** Verificar y completar los datos de un cliente usando servicios externos oficiales (SUNAT para RUC, RENIEC para DNI).

**Precondiciones:**

- El cliente tiene un DocumentoIdentidad de tipo soportado por algún servicio externo (actualmente DNI o RUC).
- La integración con las APIs SUNAT/RENIEC está operativa (configurada y habilitada).
- El usuario o el sistema interno inicia la validación (por ejemplo, un botón "Validar RUC" en la UI, o un trigger automático post-creación si la validación no se hizo en ese momento).

**Flujo Principal:**

1. El **Usuario** (o un evento interno programado) solicita validar la identidad de un cliente, pasando el `numeroDocumento` (o clientId, del cual se obtiene el doc).
2. El **Sistema** determina el tipo de documento:
  - Si es RUC, invoca la **API de SUNAT**; si es DNI, invoca la **API de RENIEC**.
3. El **Sistema** hace una llamada al servicio externo con el número de documento.
4. El **Sistema Externo (SUNAT/RENIEC)** devuelve los datos encontrados:
  - Para RUC: razón social, dirección fiscal, estado (activo/habido).
  - Para DNI: nombres, apellidos, dirección.
5. El **Sistema (GestionClientesBC)** recibe la respuesta y actualiza el objeto **Cliente** con los datos obtenidos:
  - Si la razón social estaba vacía o es genérica, la reemplaza por la razón social oficial.
  - Actualiza la dirección si estaba vacía (o podría agregar una etiqueta "Dirección fiscal").
  - Marca `FechaRegistro` de última actualización.
6. El **Sistema publica un evento** `ClienteModificado` indicando que el cliente fue completado con datos verificados.
7. El **Sistema** confirma al usuario (si fue un usuario quien solicitó) que la validación fue exitosa y muestra/actualiza en pantalla los nuevos datos.

**Flujos Alternativos:**

- *4a. API no responde:* Si el servicio externo está caído o tarda demasiado, el sistema registra un error (log) y devuelve al usuario un aviso "Servicio de verificación no disponible, intente más tarde". No se cancela al cliente, simplemente no se completan datos ahora.
- *4b. Documento no encontrado:* Si la API responde "no existe" (por ejemplo un RUC inválido o dado de baja), el sistema lanza excepción `DocumentoNoEncontradoException` o similar. Podría notificar al usuario "El RUC no figura en SUNAT" para que lo revise.
- *5a. Sin cambios:* Es posible que todos los datos ya estuvieran completos y correctos; en ese caso la API devuelve lo mismo. El sistema podría no generar evento si no hubo cambio, o generar `ClienteModificado` con `camposModificados` vacío (poco útil). En general, si no hubo alteración no se haría mucho más.

**Eventos Generados:** `ClienteModificado` (tras actualización con datos externos).

**Entidades Afectadas:** `Cliente` (campos nombre, dirección u otros actualizados).

**Diagrama de Secuencia:**



Figura 9: Flujo "Validar Identidad Externa". El sistema integra con servicios gubernamentales para enriquecer la información del cliente.

### 3.1.10 Registrar Operación en Historial

**Actor Principal:** Subsistema de Eventos (es decir, acciones desencadenadas por otros BC, sin intervención humana directa).

**Propósito:** Registrar automáticamente operaciones como facturas emitidas, pagos recibidos u otras en el historial del cliente correspondiente. Esto mantiene actualizado el perfil del cliente con sus transacciones.

#### Precondiciones:

- Se recibe un evento inbound relevante, por ejemplo:
  - ComprobanteEmitido o ComprobanteAnulado de ComprobantesElectonicosBC.
  - PagoExternoRecibido de ControlCajaBC.
- Estos eventos incluyen un clienteId que vincula la operación con un cliente en GestiónClientesBC.
- El cliente referido existe (si no, se maneja alt).

#### Flujo Principal:

1. Un **Event Handler** dentro de GestiónClientesBC recibe un evento de dominio de otro contexto (facturación o caja).
2. El **Sistema** toma el clienteId del evento y busca al cliente correspondiente en su repositorio.
3. El **Sistema** crea una nueva **OperacionCliente** con los detalles:
  - TipoOperacion derivado del tipo de evento (ej. FACTURA\_EMITIDA, FACTURA\_ANULADA, PAGO\_RECIBIDO).
  - MontoOperacion según monto del evento (total de la factura o monto del pago).
  - ReferenciaId asignando el id del comprobante o del pago original.
  - FechaOperacion tomada del evento (fecha de emisión/pago) o la actual si no viene.
4. Inserta la OperacionCliente en la lista de historial del cliente.
5. El **Sistema** publica un evento OperacionClienteRegistrada con los datos de la operación recién agregada.
6. (No hay respuesta a un actor, es asíncrono. Quizás log interno: "Historial actualizado".)

#### Flujos Alternativos:

- 2a. *Cliente no existe:* Si por alguna razón llega un evento con un clienteId que GestiónClientesBC no conoce (posiblemente error de referencia), el sistema puede **descartar el evento o enviar una alerta**. Por ejemplo, loggear "evento X no procesado: cliente no encontrado".
- 3a. *Duplicado:* Si ya había una operación con el mismo referenciaId (posiblemente se recibió evento duplicado), el sistema podría detectar y no duplicar la entrada (esto no se menciona, pero sería un guard impediente de duplicados en historial).
- *Otros eventos:* Si llegara un evento no reconocido en este contexto, simplemente se ignora.

**Eventos Generados:** OperacionClienteRegistrada .

**Entidades Afectadas:** OperacionCliente (nueva entidad creada) asociada al Cliente correspondiente.

#### Diagrama de Secuencia:



Figura 10: Flujo "Registrar Operación en Historial". GestiónClientesBC recibe eventos de otros contextos e inserta automáticamente entradas en el historial del cliente.

### 3.1.11 Consultar Historial de Operaciones

**Actor Principal:** Usuario del sistema (por ejemplo, contable o auditor que revisa actividad de un cliente) o Auditor (rol especializado).

**Propósito:** Obtener una lista filtrada de operaciones históricas de un cliente para visualizar su actividad o realizar análisis.

#### Precondiciones:

- El cliente existe.
- El actor tiene permiso de consulta (generalmente cualquier usuario que pueda ver clientes puede ver su historial; quizás ciertos roles solo lectura).
- Opcionalmente, existen filtros de búsqueda (rango de fechas, tipo de operación, estado).

#### Flujo Principal:

1. El **Usuario** solicita ver el historial de un cliente determinado, pudiendo especificar filtros como `fechaDesde`, `fechaHasta` y/o `tipoOperacion` para refinar la búsqueda.
2. El **Sistema** verifica que el cliente existe para ese usuario/tenant.
3. El **Sistema** recupera las entidades **OperacionCliente** asociadas a ese cliente que cumplan con los filtros (si se proporcionaron):
  - Filtra por fecha en el rango dado.
  - Filtra por tipoOperacion si se especificó (ej. solo pagos, solo facturas).
  - Puede ordenar descendente por fecha para presentar las más recientes primero.
4. El **Sistema** construye una **lista paginada** de resultados, especialmente importante si el historial es muy grande. Por defecto tal vez devuelve los últimos N registros.
5. El **Sistema** devuelve la lista de operaciones (con campos como fecha, tipo, monto, referencia) al usuario solicitante.
6. El **Usuario** visualiza la información. (Fuera del sistema, quizás la UI muestra en tabla o genera un reporte).

#### Flujos Alternativos:

- *2a. Cliente no existe/no accesible:* Si se consulta un cliente inválido o ajeno a su tenant, lanza `EntityNotFoundException` y no devuelve nada.
- *0a. Filtros no proporcionados:* El sistema devuelve el historial completo (posiblemente limitado por paginación).
- *3a. Sin operaciones:* Si el cliente no tiene ninguna operación en el periodo o tipo buscado, el sistema devuelve una lista vacía, posiblemente con un indicador "Sin operaciones en este criterio".
- *5a. Múltiples páginas:* Si hay más de X resultados, el sistema podría indicar un token o índice para obtener la siguiente página (paginación).
- *Errores:* No muchos esperados, ya que es consulta lectura pura.

**Eventos Generados:** Ninguno (es una operación de consulta pura, sin modificar estado).

**Entidades Afectadas:** Ninguna modificación; se leen **OperacionCliente** existentes.

#### Diagrama de Secuencia:



Figura 11: Flujo "Consultar Historial de Operaciones". El usuario solicita el historial y el sistema retorna los registros filtrados; no se generan eventos pues es lectura.

### 3.1.12 Importar Clientes desde CSV/Excel ↗

**Actor Principal:** Usuario del sistema (usualmente un administrador que quiere cargar muchos clientes).

**Propósito:** Procesar un archivo de clientes en lote para crear o actualizar múltiples registros de clientes de una sola vez.

#### Precondiciones:

- El usuario tiene un archivo CSV/Excel con columnas correctamente formateadas (ej. cabeceras estándar: tipoDoc, numDoc, nombre, email, etc.).
- Las columnas obligatorias están presentes en el archivo (DocumentoIdentidad, Nombre...).
- Formato del archivo válido (CSV delimitado correctamente o Excel con hoja con formato conocido).

#### Flujo Principal:

1. El **Usuario** carga/sube el archivo CSV o Excel a través de la interfaz.
2. El **Sistema** (GestionClientesBC, probablemente vía un servicio de importación) valida la **estructura** del archivo:
  - Verifica que existan todas las columnas requeridas (p. ej. si falta la columna "NúmeroDocumento", se aborta).
  - Opcional: valida que el archivo no exceda cierto tamaño o número de filas predefinido.
3. El **Sistema** inicia el procesamiento en lote:
  - Lee el archivo fila por fila.
  - Por cada fila (que representa un cliente o potencial cliente):
    - a. Si el **DocumentoIdentidad** ya existe en la base de clientes, entonces se **actualiza** ese cliente con los datos de la fila (equivalente a ejecutar caso de uso *Editar Cliente*).
    - b. Si no existe, se **crea** un nuevo cliente con esos datos (equivalente a *Crear Cliente*).
  - Cada fila se valida independientemente (formato de campos, etc.), aplicando las mismas reglas RN-GC que en creación manual.
  - Después de procesar cada fila con éxito, el sistema **publica el evento correspondiente:** `ClienteCreado` o `ClienteModificado` según el caso.
4. El **Sistema** acumula los resultados de cada fila (por ejemplo, contadores de creados, actualizados, errores).
5. Al finalizar todas las filas, el **Sistema** devuelve al usuario un **resumen** de la importación:
  - Número de clientes creados, número de actualizados.
  - Lista de errores encontrados (p. ej. fila 5: "RUC inválido"; fila 12: "Correo faltante").
6. El **Usuario** revisa el resumen. Si hay errores, puede corregir el archivo y reintentar o arreglar manualmente los pocos casos fallidos.

#### Flujos Alternativos:

- 2a. *Formato de archivo inválido:* Si el archivo no tiene la estructura esperada (por ejemplo, es CSV mal delimitado o falta cabeceras obligatorias), el sistema rechaza de inmediato la importación con error "Formato de archivo inválido".
- 3a. *Error en fila:* Si una fila tiene datos erróneos (ej. email mal formado), el sistema **omite esa fila** y la marca como error en el resumen. La importación continúa con las demás filas. Ningún evento de Creado/Modificado se emite para filas con error.
- 3b. *Abortar en masa:* Alternativamente, podría decidir abortar toda la importación ante el primer error, pero según doc, parece que opta por omitir y continuar ("Filas con errores → omitir y reportar").

- *3c. Cliente duplicado en archivo:* Si el mismo documento aparece en dos filas, el comportamiento puede depender: quizás la primera crea y la segunda detecta que ya existe y por tanto actualiza lo que ya se creó hace un instante (dentro del mismo proceso). Esto requiere manejo cuidadoso para que dentro del loop se considere los nuevos creaciones en la misma corrida.
- *5a. Demora en procesar:* Si son cientos/miles de filas, esto podría tardar. Posiblemente se ejecuta asíncronamente en background y el usuario recibe el resultado cuando termina. Pero en doc se asume secuencial sencillo (no se menciona asíncronía).
- *Permisos:* Solo roles admin pueden importar.

**Eventos Generados:** Por cada fila exitosa, `ClienteCreado` o `ClienteModificado`. Al final, no se genera un evento resumen (eso es solo interfaz).

**Entidades Afectadas:** múltiples `Cliente` (creados o modificados en lote), con sus subentidades si aplica.

#### Diagrama de Secuencia:



Figura 12: Flujo "Importar Clientes desde Excel". El sistema procesa cada fila, creando o actualizando clientes y emitiendo eventos correspondientes, luego envía un resumen al usuario.

#### 3.1.13 Consultar Cliente

**Actor Principal:** Usuario del sistema (por ejemplo, un vendedor que busca info del cliente) o Vendedor (rol comercial).

**Propósito:** Obtener la ficha completa de un cliente, incluyendo todos sus datos maestros, contactos, adjuntos e incluso un extracto de su historial reciente.

#### Precondiciones:

- Cliente existe y pertenece al mismo tenant del usuario.
- Usuario tiene permiso de lectura de clientes (generalmente sí).

#### Flujo Principal:

1. El **Usuario** solicita consultar un cliente específico, proporcionando el `clienteId` (o busca por documento/nombre en cuyo caso primero localizaría el id).
2. El **Sistema** busca el `Cliente` en la base de datos y verifica que existe en el contexto del usuario.
3. El **Sistema** reúne toda la información relacionada:
  - Los datos maestros del cliente (nombre, documento, tipoCliente, estado, moneda por defecto, etc.).
  - Su lista de `ContactoCliente` asociados.
  - Su lista de `AdjuntoCliente` asociados.
  - Opcionalmente, las últimas  $N$  operaciones (por ejemplo, últimos 12 meses de `OperacionCliente`) para dar un panorama reciente de actividad.
4. El **Sistema** compone un **objeto compuesto** o DTO con toda esta información estructurada (por ejemplo, un JSON con secciones "cliente", "contactos", "adjuntos", "historial").
5. El **Sistema** devuelve este objeto al **Usuario**.
6. El **Usuario** visualiza la ficha del cliente con todos los detalles integrados.

#### Flujos Alternativos:

- *2a. Cliente no encontrado:* Si el `clienteId` no existe (o no corresponde al tenant), se lanza `EntityNotFoundException` y no se devuelve ficha.

- 3a. *Sin contactos/adjuntos*: Si no tiene algunos de esos, simplemente esas listas van vacías en la respuesta.
- 3b. *Historial muy extenso*: Podría limitarlo a un periodo (como indica doc: "últimos 12 meses") para evitar exceso de datos. Si se quisiera todo el historial, tal vez se use 3.1.11 separadamente.
- Otros: No genera eventos porque es solo lectura.

**Eventos Generados:** Ninguno (consulta).

**Entidades Afectadas:** Ninguna modificada. Se leen `Cliente`, `ContactoCliente`, `AdjuntoCliente`, `OperacionCliente`.

**Diagrama de Secuencia:**



Figura 13: Flujo "Consultar Cliente". El sistema recopila todos los datos relacionados a un cliente y los retorna como un único agregado de información para la vista.

## Eventos de Dominio\_gc

GestionClientesBC emite varios **eventos de dominio** tanto internos como externos para comunicar cambios significativos en el estado de los clientes. A continuación se listan y documentan cada uno de estos eventos, incluyendo su **nombre**, **contenido (payload)** con campos y tipos principales, el **contexto que lo produce**, los **contextos que lo consumen**, y un **ejemplo de mensaje en JSON** que representa el contrato de dicho evento.

(Nota: Todos los eventos son mensajes inmutables que indican que algo ya ocurrió en el dominio de clientes. Son publicados después de que la transacción del caso de uso se completa satisfactoriamente, garantizando que los datos del evento reflejan el nuevo estado persistido.)

### 4.1 Detalle de cada Evento de Dominio ↗

#### Evento: ClienteCreado ↗

- **Descripción:** Este evento se dispara cuando un usuario del sistema crea un nuevo cliente en Factura Fácil. Marca el **inicio del ciclo de vida** de un cliente dentro de la plataforma.
- **Productor:** GestiónClientesBC (agregado Cliente) tras la finalización exitosa del caso de uso *Crear Cliente* (ya sea manual o auto-creación). Solo se publica una vez por cliente.
- **Consumidores Potenciales:**
  - **IndicadoresNegocioBC:** Consumir `ClienteCreado` para incluir el nuevo cliente en dashboards y segmentaciones de negocio (por ejemplo, incrementar contadores de clientes activos, actualizar informes).
  - **UI/Frontend:** Podría suscribirse para actualizar la lista de clientes en tiempo real en la aplicación (mostrar automáticamente el nuevo cliente).
  - **Otros contextos internos:** Quizá un módulo de notificaciones para alertar al equipo comercial de un cliente nuevo.
- **Payload (Datos contenidos):**
  - `clienteId` : UUID que identifica al cliente recién creado.
  - `documentoIdentidad` : Objeto de valor con tipo y número de documento (por ejemplo, {"tipo": "RUC", "numero": "20601234571"}).
  - `nombreCompleto` o `razonSocial` : Cadena con el nombre o razón social del cliente.
  - `correo` : Email principal del cliente (string, puede ser null si no se registró).
  - `telefono` : Teléfono principal del cliente (string o número).
  - `tipoCliente` : Categoría asignada (e.g. "MINORISTA").
  - (Se podrían incluir otros datos básicos si estuvieran disponibles, como dirección principal).
- **Contrato de mensaje (JSON Ejemplo):**

```
1 {
2 "event": "ClienteCreado",
3 "clienteId": "a1f5c830-5e9b-11ec-90d6-0242ac120003",
4 "documentoIdentidad": { "tipo": "RUC", "numero": "20601234571" },
5 "razonSocial": "IMPORTACIONES PERU SAC",
```

```
6 "correo": "contacto@importaciones.pe",
7 "telefono": "012233445",
8 "tipoCliente": "MAYORISTA",
9 "fechaCreacion": "2025-07-04T18:22:00Z"
10 }
```

(En este ejemplo, un cliente empresa con RUC fue creado. El evento incluye sus datos principales. Consumidores como IndicadoresNegocioBC usarán esta información para agregar el cliente a análisis, mientras que la UI podría, por ejemplo, mostrar una notificación "Cliente 'IMPORTACIONES PERU SAC' registrado exitosamente".)

#### Evento: ClienteModificado

- **Descripción:** Notifica que uno o varios datos de un cliente existente han sido cambiados. Sirve para auditar y reflejar **cambios en la información** del cliente a través del tiempo.
- **Productor:** GestiónClientesBC, típicamente en los casos de uso *Editar Cliente*, *Asignar Tipo de Cliente*, *Gestionar Contacto (Editar/Eliminar)*, *Gestionar Adjuntos (Eliminar)*, o *Validar Identidad Externa*. Es decir, cualquier operación que actualice la ficha del cliente produce este evento.
- **Consumidores Potenciales:**
  - **IndicadoresNegocioBC:** Actualiza reportes o triggers de alertas basados en cambios de datos. Por ejemplo, si cambió el segmento (tipoCliente) de un cliente importante, notificar área de ventas.
  - **UI/Frontend:** Si un usuario está viendo la ficha, podría refrescar los datos en tiempo real al recibir ClienteModificado.
  - **ListaPreciosBC:** Específicamente consume este evento para detectar cambios de TipoCliente o EstadoCliente y ajustar listas de precios asociadas.
- **Payload:**
  - **clienteId** : UUID del cliente modificado.
  - **camposModificados** : Lista de objetos indicando cada campo cambiado, incluyendo el nombre del campo, su valor anterior y su valor nuevo. Por ejemplo:

```
1 "camposModificados":
2 [
3 {
4 "campo": "correo",
5 "antes": "old@mail.com",
6 "despues": "new@mail.com"
7 },
8 {
9 "campo": "tipoCliente",
10 "antes": "MINORISTA",
11 "despues": "MAYORISTA"
12 }
13 ]
```

- **fechaCambio** : Timestamp de cuándo se realizó la modificación.
- (Opcionalmente se podría incluir quién lo modificó si se agrega contexto, aunque usualmente no en el evento de dominio puro.)
- **Contrato JSON Ejemplo:**

```

1 {
2   "camposModificados": [
3     {
4       "antes": "contacto@importaciones.pe",
5       "campo": "correo",
6       "despues": "ventas@importaciones.pe"
7     },
8     {
9       "antes": "MAYORISTA",
10      "campo": "tipoCliente",
11      "despues": "DISTRIBUIDOR"
12    }
13  ],
14  "clienteId": "a1f5c830-5e9b-11ec-90d6-0242ac120003",
15  "event": "ClienteModificado",
16  "fechaCambio": "2025-08-10T09:30:15Z"
17 }
18

```

*Interpretación:* El cliente cambió su correo principal y su categoría a DISTRIBUIDOR. IndicadoresNegocioBC al recibir esto podría actualizar el segmento del cliente en sus datos y la UI podría reflejar el nuevo correo en la vista si está abierta.

- **Observaciones:** Este evento facilita la auditoría de todos los cambios en clientes. Cada consumidor puede decidir si utiliza la lista de cambios o simplemente detectar que hubo una modificación.

#### Evento: ClienteDeshabilitado

- **Descripción:** Indica que un cliente ha sido marcado como **inactivo** en el sistema. A partir de este evento, el cliente no debería participar en nuevas operaciones comerciales.
- **Productor:** GestiónClientesBC al completar el caso de uso *Deshabilitar Cliente* (cambio de estado a INACTIVO).
- **Consumidores Potenciales:**
  - **IndicadoresNegocioBC:** Excluye al cliente inactivo de sus métricas (por ejemplo, al calcular clientes activos totales).
  - **ComprobantesElectrónicosBC:** Puede suscribirse para evitar emitir futuras facturas a este cliente. Por ejemplo, cuando un usuario intente facturar, Facturación podría verificar un caché o consulta rápida del estado del cliente (o directamente suscribirse a este evento para actualizar su base de datos local).
  - **Otros:** Módulos de CRM o atención al cliente podrían actualizar estatus (inactivo, ya no enviar promociones, etc.).
- **Payload:**
  - `clienteId` : UUID del cliente deshabilitado.
  - `motivoDeshabilitacion` : Texto con la razón del cambio (podría ser null si no se proporcionó).
  - `fechaCambio` : Timestamp de cuándo se realizó la deshabilitación.
- **Contrato JSON Ejemplo:**

```

1 {
2   "clienteId": "d4e6f410-7107-11ec-8e32-0242ac130003",
3   "event": "ClienteDeshabilitado",
4   "fechaCambio": "2025-09-01T00:00:00Z",
5   "motivoDeshabilitacion": "Cliente inactivo por falta de pago"
6 }

```

*Significado:* El cliente con ese ID fue puesto inactivo el 1 de septiembre de 2025 debido a falta de pago.

- Facturación electrónica podría al recibir esto bloquear nuevas ventas a ese cliente (como sugiere el doc).
- Indicadores lo sacará de cálculos de crecimiento de clientes.
- **Importante:** Desde este evento en adelante, *el cliente queda bloqueado para toda operación nueva*, lo cual los consumidores deben respetar.

#### Evento: OperacionClienteRegistrada

- **Descripción:** Señala que se añadió una nueva operación (transacción histórica) al historial de un cliente. Representa hechos como "El cliente X tuvo una factura emitida Y" o "recibió un pago Z".
- **Productor:** GestiónClientesBC, específicamente el handler interno del caso de uso *Registrar Operación en Historial* que se activa por eventos de otros contextos (facturación, caja).
- **Consumidores Potenciales:**
  - **IndicadoresNegocioBC:** Actualiza sus dashboard de ventas, cobranzas y cuentas por cobrar con esta nueva operación. Por ejemplo, incrementa el total vendido del mes, reduce saldo pendiente, recalcula promedio de pago, etc.
  - **Módulos internos de Auditoría/Contabilidad:** Pueden conciliar las operaciones registradas con las de facturación/caja, ya que este evento confirma que la operación fue registrada en el perfil de cliente.
  - **Otros sistemas financieros externos:** Si existieran integraciones, podrían usarlo para actualizar su representación del estado de cliente.
- **Payload:**
  - `clienteId` : UUID del cliente relacionado.
  - `operacionId` : Identificador único de la operación en el contexto de clientes (podría ser igual al de la factura/pago o uno generado).
  - `tipoOperacion` : Un string o código representando el tipo de operación histórica (por ejemplo, "FACTURA\_EMITIDA", "PAGO\_RECIBIDO").
  - `montoOperacion` : Monto involucrado (decimal). Podría incluir moneda si fuera relevante, pero generalmente la moneda se deduce del comprobante externo; es posible que se normalice a la moneda base del sistema.
  - `referenciaId` : Identificador del comprobante o pago en su sistema de origen (por ejemplo, invoiceId en facturación o paymentId en caja).
  - `fechaOperacion` : Fecha/hora en que ocurrió la operación.

#### • Contrato JSON Ejemplo:

```
1 {
2   "clienteId": "f7a1b780-82f3-11ec-a8a3-0242ac140004",
3   "event": "OperacionClienteRegistrada",
4   "fechaOperacion": "2025-07-04T15:30:00Z",
5   "moneda": "PEN",
6   "montoOperacion": 1250.50,
7   "operacionId": "INV-100045",
8   "referenciaId": "FAC-2025-00012345",
9   "tipoOperacion": "FACTURA_EMITIDA"
10 }
```

**Interpretación:** El cliente con ID dado tuvo una **FACTURA\_EMITIDA** registrada en su historial, por S/1250.50, con referencia FAC-2025-00012345 (ID del comprobante en ComprobantesElectronicosBC) a las 3:30pm del 4 de julio de 2025.

- IndicadoresNegocioBC recibirá esto y sumará 1250.50 a la métrica de ventas del día, por ejemplo.
- Un sistema contable podría marcar esa factura como contabilizada en el ledger de cliente.
- **Observaciones:** Este evento es crítico para mantener sincronizados los saldos y reportes financieros, y debe procesarse de manera confiable (idealmente con garantías de entrega, ya que si se pierde uno, los datos podrían descuadrarse).

#### Evento: ContactoAgregado

- **Descripción:** Informa que se añadió un nuevo medio de contacto al cliente. Este evento tiene un impacto más leve, generalmente para interfaces de usuario o integraciones de comunicaciones.
- **Productor:** GestiónClientesBC, caso de uso *Gestionar Contacto* (sub-flujo agregar).
- **Consumidores Potenciales:**
  - **UI/Notificaciones:** Podría notificar al usuario administrador que se agregó un contacto exitosamente, o refrescar la lista de contactos en la vista del cliente.
  - **Sistemas de mailing/CRM:** Si hay un servicio que centraliza contactos, podría escuchar este evento para sincronizar (por ej., agregar ese email a la lista de contactos en un CRM externo).
- **Payload:**
  - `clienteId` : UUID del cliente al que se agregó el contacto.
  - `contactoId` : Identificador único del nuevo contacto.
  - `tipoContacto` : Tipo del contacto agregado (por ejemplo "EMAIL\_SECUNDARIO", "TELEFONO\_SECUNDARIO", "DIRECCION").
  - `valorContacto` : El valor específico añadido (el email, el número de teléfono, etc.).
- **Contrato JSON Ejemplo:**

```

1 {
2   "clienteId": "c2b4f7e0-90ab-11ec-bb12-0242ac150005",
3   "contactoId": "EMAIL-002",
4   "event": "ContactoAgregado",
5   "tipoContacto": "EMAIL_SECUNDARIO",
6   "valorContacto": "soporte@empresa.com"
7 }
```

**Interpretación:** Al cliente con id `c2b4f7e0...` se le agregó un nuevo contacto con id "EMAIL-002" del tipo EMAIL\_SECUNDARIO con dirección [soporte@empresa.com](mailto:soporte@empresa.com).

- Un front-end al recibirla podría inmediatamente mostrar ese correo en la UI del cliente.
- No tiene implicancias en otros contextos de negocio (no afecta facturación ni precios, etc.), es más bien informativo.
- **Observaciones:** Este evento "*Mejora la riqueza del perfil de cliente para comunicaciones futuras*". Por ejemplo, un módulo de envío de boletines podría acumular estos contactos para futuras campañas.

## Evento: AdjuntoAgregado

- **Descripción:** Publica que se ha **adjuntado un documento o imagen** al perfil de un cliente. Permite a otros sistemas conocer la existencia de nuevos documentos asociados al cliente.
- **Productor:** GestionClientesBC, caso de uso *Gestionar Adjuntos* (sub-flujo agregar).
- **Consumidores Potenciales:**
  - **Sistema de Almacenamiento/Documentos:** Podría escuchar para sincronizar con un repositorio externo o generar backups del archivo subido.
  - **UI:** Para habilitar la descarga inmediata del nuevo adjunto en la interfaz (por ejemplo, mostrar un link de descarga).
  - **Auditoría:** Un sistema de compliance podría registrar que cierto documento (DNI, contrato) fue subido al perfil de tal cliente en tal fecha, para trazabilidad legal.
- **Payload:**
  - `clienteId` : UUID del cliente.
  - `adjuntoId` : Identificador del adjunto agregado (podría ser un UUID o un código generado).
  - `tipoAdjunto` : Categoría del adjunto (e.g. "CONTRATO", "FOTO", "DOCUMENTO\_IDENTIDAD", "OTRO").
  - `nombreArchivo` : Nombre del archivo adjunto (ej. "Contrato\_ABC.pdf").
  - `ruta` : Ruta o URL donde está almacenado el archivo (puede ser interna o en un servicio externo tipo S3).
  - `fechaSubida` : Timestamp de cuándo se subió.
- **Contrato JSON Ejemplo:**

```
1 {  
2   "adjuntoId": "ATT-20250704-001",  
3   "clienteId": "c2b4f7e0-90ab-11ec-bb12-0242ac150005",  
4   "event": "AdjuntoAgregado",  
5   "fechaSubida": "2025-07-04T18:00:00Z",  
6   "nombreArchivo": "ContratoServicio_2025.pdf",  
7   "ruta": "https://files.facturafacil.com/adjuntos/ContratoServicio_2025.pdf",  
8   "tipoAdjunto": "CONTRATO"  
9 }  
10
```

*Interpretación:* Se adjuntó un contrato (PDF) al cliente especificado. La ruta muestra que está alojado en un servidor de archivos.

- Un módulo de gestión documental externo podría ver este evento y guardar meta-information del archivo.
- La UI ahora puede mostrar un ícono de contrato disponible para ese cliente.
- **Observaciones:** Este evento garantiza **sopporte documental y respaldo para auditorías**. Es decir, se puede asegurar que documentos importantes quedan registrados. No suele afectar procesos transaccionales directamente, pero asegura que en cualquier punto del tiempo se sepa que el cliente tiene ciertos documentos adjuntos.

## 4.2 Resumen de Eventos (Internos vs Externos)

Todos los eventos listados arriba (*ClienteCreado*, *ClienteModificado*, *ClienteDeshabilitado*, *OperacionClienteRegistrada*, *ContactoAgregado*, *AdjuntoAgregado* y eventualmente *ClienteEliminado*) son **eventos de dominio outbound** de

GestionClientesBC. En la documentación original, se mapearon consumidores internos y externos en un cuadro.

Resumiendo relaciones:

- **Eventos Publicados por GestionClientesBC:**

EventoDestino (Consumidor)Propósito / Uso**ClienteCreado**IndicadoresNegocioBCIncluir cliente en dashboards y segmentaciones. (También UI, etc.)**ClienteModificado**IndicadoresNegocioBC, ListaPreciosBCRefrescar datos en reportes y ajustar segmentaciones o precios.**ClienteDeshabilitado**IndicadoresNegocioBC, ComprobantesBCExcluir cliente inactivo de análisis métricas y bloquear nuevas facturaciones.**ClienteEliminado**(*No estaba explícito, pero similar a deshabilitado:*) Podría ser auditado en Indicadores (baja definitiva).**OperacionClienteRegistrada**IndicadoresNegocioBC, Contabilidad/AuditoríaActualizar dashboards de ventas y cuentas por cobrar; conciliar movimientos financieros.**ContactoAgregado**UT / NotificacionesNotificar al usuario y mostrar nuevo contacto en perfil.**AdjuntoAgregado**Sistemas de almacenamiento, UISincronizar documentos externos, habilitar descarga en interfaz.

- **Eventos Consumidos por GestionClientesBC (Inbound):**

Evento (desde otro BC)OrigenUso en GestionClientesBC**ComprobanteEmitido**ComprobantesElectonicosBCRegistrar en historial (FACTURA\_EMITIDA), auto-crear cliente si no existe.**ComprobanteAnulado**ComprobantesElectonicosBCRegistrar en historial (FACTURA\_ANULADA).**PagoExternoRecibido**ControlCajaBCRegistrar en historial (PAGO\_RECIBIDO).(*Otros potenciales*):(*e.g., NotaCreditoEmitida en futuro, etc.*)

Estos eventos inbound llegan vía el bus/eventos y desencadenan los casos de uso correspondientes (auto-creación o registro de operación).

En conclusión, los eventos de dominio son esenciales para la **integración desacoplada** de GestionClientesBC con el resto del sistema, permitiendo una arquitectura orientada a eventos donde cada contexto reacciona a cambios en los demás de forma asíncrona y robusta.

# Restricciones y Reglas de Negocio\_gc

## 1. Contexto

GestionClientesBC opera bajo varias **restricciones e invariantes** que **no pueden ser violados** en ninguna circunstancia, así como **reglas de negocio** que definen comportamientos del sistema en situaciones específicas. Estas reglas aseguran la integridad de los datos de clientes y alinean el sistema con las normativas vigentes. A continuación se detallan las reglas más relevantes, citando su código de identificación (RN-GC-XXX) cuando corresponde y referencias del documento:

- **RN-GC-001 – Documento válido y único:** Cada cliente debe tener un **Documento de Identidad único** en el sistema y con formato correcto. Esto implica:
  - *Unicidad:* No puede haber dos clientes con el mismo número de DNI, RUC, etc. Al intentar crear o actualizar, se verifica en la base de datos que ningún otro cliente activo tenga ese documento.
  - *Formato:* El `tipoDocumento` define cómo debe ser el `numeroDocumento`. Por ejemplo, DNI son 8 dígitos numéricos; RUC son 11 dígitos numéricos con un dígito verificador válido; Pasaporte puede incluir letras/números con longitud variable, etc. Si el formato no corresponde, la operación es rechazada (Ej: un RUC de 10 dígitos daría *DocumentoInvalidoException*).
  - *Campos obligatorios:* Adicionalmente, esta regla cubre que **nombre/razón social y documento son campos obligatorios** para un cliente. No se permitirá guardar un cliente sin identificarlo con al menos un nombre y su DNI/RUC correspondiente.
  - **Ejemplo:** Si un usuario intenta crear un cliente sin ingresar RUC, el sistema no lo permitirá (error: "DocumentoIdentidad es obligatorio"). Si intenta duplicar DNI ya existente, error "Ya existe un cliente con este DNI".
- **RN-GC-002 – Formato de correo electrónico:** Cada dirección de correo registrada en un cliente (sea en sus datos principales o contactos secundarios) debe cumplir el **estándar RFC-5322** de emails. Esto significa que debe tener formato "usuario@dominio" válido (no puede faltar la '@', ni caracteres inválidos).
  - El sistema valida esto en la creación/edición de cliente (`correo principal`) y al agregar/editar un contacto de tipo email.
  - **Ejemplo:** "cliente@@dominio.com" o "cliente@dominio" serían rechazados inmediatamente con error de validación de email.
- **RN-GC-003 – Estado INACTIVO sin operaciones:** Un cliente en estado INACTIVO **no puede iniciar nuevas transacciones**. Esta es una política operativa:
  - La propia GestionClientesBC no lo impide directamente (no hay "transacción" dentro del contexto de cliente), pero comunica esta restricción a otros contextos. ComprobantesElectonicosBC, al intentar facturar, debe consultar el estado del cliente; si está INACTIVO, debe bloquear la emisión (posiblemente devolviendo error al usuario "Cliente Inactivo, no se puede facturar").
  - Asimismo, al intentar registrar un pago en ControlCajaBC para un cliente inactivo, se podría avisar al usuario. Esto depende de implementaciones en esos contextos.
  - GestionClientesBC implementa RN-GC-003 al asegurarse de emitir el evento ClienteDeshabilitado y mantener el flag, y en sus propios casos de uso: por ejemplo, podría prohibir *Editar Cliente* financiero si

inactivo, pero usualmente solo transacciones se bloquean.

- **Ejemplo:** Si un cliente fue deshabilitado por morosidad, el vendedor no podrá seleccionarlo en el módulo de ventas (el sistema lo excluirá o avisará). Indicado por: "*Estado INACTIVO bloquea nuevas transacciones*".
- **RN-GC-004 – Auto-creación siempre ACTIVO:** Cuando el sistema auto-crea un cliente a partir de un evento de facturación, debe asignársele siempre `EstadoCliente = ACTIVO`.
  - Esto garantiza que la factura que provocó la auto-creación pueda procesarse sin impedimentos, y evita inconsistencias (un cliente recién creado no tendría por qué estar inactivo inmediatamente).
  - El código fuerza este valor en el flujo de auto-creación.
  - **Ejemplo:** Al recibir un `ComprobanteEmitido` con un RUC desconocido, se crea el cliente. Aun si la empresa tiene política de aprobar antes de activar, esa política no aplica aquí: RN-GC-004 dice que es ACTIVO de entrada, luego si hace falta podría manualmente deshabilitarlo.
- **RN-GC-005 – Coherencia de fechas en historial:** Cada registro de operación en el historial (`OperacionCliente`) debe tener `fechaOperacion >= FechaRegistro del cliente`.
  - Esto es una integridad referencial temporal: no se puede asignar una factura de 2020 a un cliente creado en 2021, al menos no sin ajustar `fechaRegistro` o considerar que el cliente fue creado "retroactivamente".
  - En la práctica, `FechaRegistro` suele ser la de creación, excepto si se mantuvo historial migrado, pero RN-GC-005 asegura que al insertar `OperacionCliente` se haga una verificación (y potencialmente ajusten `FechaRegistro` si se importan históricos antiguos).
  - **Ejemplo:** Si se importa retrospectivamente facturas del 2020 para un cliente creado hoy, el sistema debería o bien rechazar esas operaciones (violación RN-GC-005) o actualizar internamente `FechaRegistro` del cliente al 2020 para englobar su verdadera antigüedad. Lo más sencillo es no permitirlo: ergo, cualquier `OperacionCliente` con fecha anterior a la creación del cliente se considera inválida a menos que se tomen medidas especiales.
- **RN-GC-006 – Eliminación condicionada sin pendientes:** Para ejecutar la eliminación lógica de un cliente, se deben cumplir dos condiciones:
  - a. Que el cliente **no tenga operaciones pendientes** (por pagar, o procesos abiertos) ni saldos por cobrar.  
Esto suele implicar revisar que no existan facturas impagadas asociadas. `GestionClientesBC` por sí mismo no conoce el estado de pago, así que depende de una verificación con otros contextos (como en `Deshabilitar`). Para eliminar, podría aplicarse la misma lógica pero más estricta: no solo pendientes actuales, sino ninguna operación histórica que impida borrarlo por normativa (p. ej. facturas en el último año).
  - b. Realizar la acción como **soft-delete**, es decir, no borrar físicamente el registro sino marcarlo eliminado.
    - Esta regla evita que se pierda información necesaria para auditoría o legal (por ejemplo, no eliminar un cliente mientras existan facturas legales a su nombre, ya que deben conservarse ~5 años).
    - **Ejemplo:** Un usuario intenta eliminar a "Cliente XYZ" pero este tiene facturas registradas el mes pasado. El sistema no lo permitirá (RN-GC-006 falla: existen operaciones, ergo "No se puede eliminar cliente con operaciones activas"). Si todas sus facturas están cobradas y pasaron X años, recién ahí se permitiría (depende políticas, pero la regla base es que no haya pendientes).
- **Restricción de Adjuntos (sin código RN explícito):** Los adjuntos de cliente tienen restricciones técnicas:
  - **Tamaño:** Máximo 5 MB por archivo.

- **Tipo de archivo:** Solo formatos PDF, JPG o PNG permitidos.
- Estas restricciones se aplican en el caso de uso Gestión de Adjuntos. Si un archivo no cumple, la operación de adjuntar falla (p. ej. tratando de subir un .exe de 1 MB se rechaza por tipo; un .pdf de 10 MB se rechaza por tamaño).
- **Ejemplo:** Usuario intenta subir "backup.zip" de 1 MB → error "Tipo de archivo no permitido". Intenta subir "foto.png" de 7 MB → error "Archivo excede tamaño máximo".
- **Permisos y Roles:** Aunque no listadas como RN específicas en doc, hay implícitas políticas de autorización:
  - Solo usuarios con rol adecuado pueden crear/editar/deshabilitar clientes. En casos de uso se mencionó, por ejemplo, precondición "Usuario autenticado con permiso GestionClientes.Crear".
  - Roles posibles (según sección 12.2 pedirá más detalle): Rol\_Administrador puede todo, Rol\_Asistente tal vez solo ver, etc. Esto se valida antes de ejecutar los flujos.
  - Estas políticas no son del dominio puro, sino de aplicación/infraestructura (ver sección 12), pero las mencionamos para completitud.

**Referencias Normativas Externas:** Algunas de estas reglas están respaldadas por normativas:

- Formato DNI/RUC: definido por RENIEC y SUNAT respectivamente.
- Unicidad de RUC: un RUC identifica legalmente a una empresa, duplicarlo generaría conflictos fiscales.
- RFC-5322: es un estándar IETF para correos (la referencia se hizo explícita en doc).
- Retención de datos: leyes locales pueden exigir no eliminar completamente datos de clientes relacionados a comprobantes (ej. en algunos países, los comprobantes deben conservarse 5 años, por lo que se sugiere soft-delete en vez de borrar).
- Tamaño de archivos: 5MB es más bien una decisión técnica para no saturar almacenamiento ni transferencias, no una ley, pero es política interna.

#### Cumplimiento de las Reglas:

- GestionClientesBC implementa estas restricciones en diferentes capas:
  - En el modelo de dominio (VO y entidades) se implementan validaciones de formato (DNI, email, etc.).
  - En los servicios de dominio o la capa de aplicación, se checan condiciones de negocio (unicidad contra repositorio, permisos de usuario, etc.).
  - Antes de cambiar estados o eliminar, se consultan otros contextos si es necesario (pendientes de pago, etc.).
- Además, cada RN tiene código para trazabilidad. Por ejemplo, en la documentación se listan RN-GC-001 a RN-GC-006 con sus definiciones, lo cual permite al equipo referirse a ellas fácilmente en discusiones y documentación de pruebas.

En la práctica, estas reglas garantizan que **los datos de clientes sean confiables y el sistema se comporte correctamente** ante las operaciones permitidas. Cualquier intento de violar una restricción resulta en excepciones o mensajes de error claros, manteniendo la integridad del dominio de clientes en Factura Fácil.

# Lenguaje Ubicuo\_gc

## 1. ¿Qué es el Lenguaje Ubicuo?

El **Lenguaje Ubicuo** (Ubiquitous Language) es el vocabulario compartido entre desarrolladores, expertos de dominio y stakeholders. Sirve para:

- **Alinear términos** técnicos y de negocio en toda la documentación y el código.
- **Reducir ambigüedades**, garantizando que cada concepto del dominio tenga una única definición.
- **Conectar** entidades, procesos y reglas de negocio mediante nomenclatura común.

## 2. Glosario de Términos

Término	Definición	Notas / Contexto
Cliente	Agregado raíz con datos maestros, contactos, adjuntos e historial de operaciones.	Punto de entrada para CRUD de clientes.
DocumentoIdentidad	VO que une <b>TipoDocumento</b> + número y valida longitud y dígito verificador.	Usado en <b>Cliente</b> para identificación y validación fiscal.
TipoDocumento	Enum de tipos de documento: <b>DNI</b> , <b>RUC</b> , <b>Pasaporte</b> , <b>CarnéExtranjería</b> , etc.	Define reglas de validación y longitud.
TipoCliente	Enum para clasificar clientes ( <b>Mayorista</b> , <b>Minorista</b> , <b>Distribuidor</b> , etc.).	Influye en políticas de precio y segmentación en <b>ListaPreciosBC</b> .
EstadoCliente	Enum de estados: <b>Activo</b> , <b>Inactivo</b> , <b>Eliminado</b> .	Controla si un cliente puede participar en transacciones.
FechaRegistro	VO Timestamp que marca creación o última actualización.	Se almacena en <b>Cliente</b> para auditoría.

ContactoCliente	Entidad que representa medios de contacto (correo, teléfono, dirección).	Permite múltiples contactos por cliente.
AdjuntoCliente	Entidad que almacena documentos o imágenes asociadas a un cliente.	Tipos de adjunto: Contrato , Foto , DocumentoIdentidad , Otro .
OperacionCliente	Entidad que registra transacciones históricas (facturas, pagos, anulación).	Utilizada para generar informes y saldos.
TipoOperacion	VO que clasifica operaciones: FacturaEmitida , CotizacionEmitida , PagoRecibido , FacturaAnulada .	Influence reporting and event handling.

### 3. Agrupaciones Temáticas (opcional)

#### 3.1 Términos de Identificación y Validación

Término	Definición	Notas / Contexto
DocumentoIdentidad	Composición de tipo y número de documento con validación integrada.	Necesario antes de crear o editar un cliente.
TipoDocumento	Categoría de documento que determina validación y formato.	Disponible para selección en formularios.
ValidacionIdentidad	Servicio de dominio que consulta APIs externas para verificar y completar datos de cliente.	Integrado con SUNAT/RENIEC.

#### 3.2 Términos de Gestión de Cliente

Término	Definición	Notas / Contexto
Cliente	Agregado raíz que orquesta subentidades y VOs.	Núcleo de GestionClientesBC.

ContactoCliente	Medio de contacto secundario.	Gestor de comunicaciones.
AdjuntoCliente	Documento o imagen de respaldo.	Soporte documental y auditoría.
OperacionCliente	Historial de acciones relacionadas con un cliente.	Fuente de métricas y reportes de cuentas por cobrar.

#### 4. Términos Heredados de Otros Contextos (opcional) ☀

Término	Definición en este BC	Origen / Relación
ComprobanteEmitido	Evento de facturación que incluye clienteId y totales.	Inbound de <b>ComprobantesElectronico sBC</b> .
ComprobanteAnulado	Evento que notifica anulación de comprobante.	Inbound de <b>ComprobantesElectronico sBC</b> .
PagoExternoRecibido	Evento de registro de pago asociado a un cliente.	Inbound de <b>ControlCajaBC</b> .
ClienteCreado	Evento outbound generado al crear cliente.	Outbound de <b>GestionClientesBC</b> , consumido por indicadores.
ClienteModificado	Evento outbound de actualización de datos de cliente.	Outbound de <b>GestionClientesBC</b> .

#### Vocabulario Compartido ☀

- **Comandos (Commands):** órdenes que el usuario ejecuta sobre el sistema. Ejemplos:
  - RegistrarCliente o CrearClienteCommand : para dar de alta un nuevo cliente con datos como nombre, NIT, dirección, teléfono, etc.
  - ActualizarCliente o ModificarClienteCommand : para cambiar información de un cliente existente (nombre, correo, categoría, etc.).
  - EliminarCliente o DarBajaClienteCommand : para marcar un cliente como inactivo o borrarlo del sistema.
  - AsignarCategoriaCliente , AsociarContactoCliente , etc., para casos de uso específicos (p.ej. asignar un representante legal).
- **Eventos (Domain Events):** notificaciones de algo ocurrido en el dominio. Ejemplos:

- `ClienteRegistrado` o `ClienteCreado`: disparado después de procesar el comando de alta de cliente. Incluye el ID generado y los datos del cliente.
- `ClienteActualizado`: cuando se confirma que se modificó el cliente.
- `ClienteEliminado` o `ClienteInactivado`: cuando un cliente es dado de baja.
- Otros eventos derivados, como `ContactoAsociadoACliente`, `DireccionAgregada`, dependiendo del modelo. Estos eventos capturan los cambios y pueden desencadenar acciones en el mismo BC u otros.
- **Entidades (Entities)**: objetos con identidad propia que evolucionan a lo largo del tiempo. En nuestro contexto, la raíz de agregado principal es:
  - **Cliente**: representa al cliente (persona natural o jurídica). Tiene un identificador único (p.ej. UUID) y atributos de negocio: nombre o razón social, tipo de identificación (NIT, cédula, etc.), número de documento, dirección principal, correo electrónico, teléfono, categoría de cliente, etc. Cada `Cliente` mantiene sus invariantes internas. Por ejemplo, un cliente puede tener varios contactos, direcciones secundarias o representantes (entidades hijas) que pertenecen al agregado. Como dice Programadores10x, “un `Cliente` en un sistema de comercio electrónico es una entidad” con identificador único.
- **Objetos de Valor (Value Objects)**: objetos inmutables que describen atributos o conceptos de negocio sin identidad propia. Ejemplos en gestión de clientes:
  - **Dirección**: compuesta por calle, número, ciudad, código postal, etc. Es inmutable: si cambia algún componente se crea una nueva instancia en lugar de modificarla.
  - **CorreoElectrónico, Teléfono, DocumentoIdentidad** (que incluye tipo y número), **CategoríaCliente**, etc. Cada uno encapsula reglas simples (p.ej. formato válido de e-mail o de teléfono) y se usan dentro de `Cliente`. De este modo, la entidad `Cliente` trabaja con VOs para atributos ricos pero sin identidad.

### Ejemplos de uso en conversaciones

En las conversaciones con expertos de dominio se usa este vocabulario para alinear significados. Por ejemplo:

- **Usuario**: “Necesitamos dar de alta un nuevo cliente y asignarle su NIT y dirección.”
- Desarrollador**: “Entonces usaremos el comando `RegistrarCliente`, que incluye campos como `nombre`, `tipoIdentificacion`, `numeroDocumento`, `correo`, `direccion`. Al ejecutarlo, el sistema emitirá el evento `ClienteRegistrado` con esos datos”
- **Usuario**: “¿Qué pasa si el cliente cambia de dirección después?”
- Desarrollador**: “En ese caso se usa `ActualizarCliente` especificando el nuevo valor de `direccion`. Esto genera el evento `ClienteActualizado`.”
- **Usuario**: “En el negocio llamamos a eso ‘dar de baja’ a un cliente cuando se va, ¿qué término usaremos en el sistema?”
- Desarrollador**: “Adoptaremos el mismo término: el comando `EliminarCliente` (o `DarBajaCliente`) marcará el cliente como inactivo. Esto publica el evento `ClienteEliminado`. El uso del mismo nombre por el equipo de negocio y en el código asegura que todos entiendan el mismo concepto.”

# Servicios de Aplicación\_gc

## 7.1 Definición y función

Los **Application Services** (Servicios de Aplicación) son clases de la capa de aplicación que orquestan los casos de uso, coordinando repositorios, servicios de dominio y manejo de transacciones. Cada caso de uso relevante en GestiónClientesBC (p.ej. registrar o actualizar un cliente) tiene un servicio encargado. Por ejemplo, podríamos definir `ClienteApplicationService` con métodos como `crearCliente()`, `actualizarCliente()`, etc. Estos servicios:

- Reciben comandos o DTOs desde la capa externa (APIs).
- Validan reglas de aplicación (por ejemplo, verificar que el NIT no esté duplicado usando un repositorio).
- Invocan objetos del dominio (agregados, factories, domain services) para ejecutar la lógica principal.
- Persisten agregados mediante los repositorios (Repository Pattern) en una sola transacción.
- Publican eventos de dominio o integración según corresponda.

Según Navarrete, “la orquestación de repositorios, gateways y eventos corresponde a los Application Services, porque representan flujos específicos de la aplicación”. Esto mantiene la lógica de dominio pura (en las entidades) y delega en los servicios de aplicación todo lo demás (coordinación técnica, transacciones, conversión de datos).

## 7.2 Interfaces y contratos

Los Application Services exponen interfaces claras (contratos) que definen sus operaciones. Por ejemplo:

```
1 public interface GestionClientesService {  
2     ClienteResponse crearCliente(CrearClienteRequest request);  
3     ClienteResponse actualizarCliente(ActualizarClienteRequest request);  
4     void eliminarCliente(UUID clienteId);  
5     // Otros métodos según casos de uso...  
6 }
```

Aquí `CrearClienteRequest` y `ActualizarClienteRequest` son DTOs (Data Transfer Objects) que contienen los datos necesarios. Los DTOs forman parte de la capa de aplicación y definen los contratos de entrada y salida. Son estructuras simples sin lógica de negocio. Después de procesar, el servicio devuelve `ClienteResponse` (otro DTO) con los datos del cliente resultante (p.ej. ID generado, estado). De esta forma se separa el modelo de dominio interno de los formatos de intercambio con la capa de presentación o API.

## 7.3 Ejemplos de métodos

A continuación un ejemplo de firma de método y su flujo típico:

Método	Parámetros	Respuesta
<code>crearCliente()</code>	<code>CrearClienteRequest</code> <code>{ nombre, nit, ... }</code>	<code>ClienteResponse {</code> <code>id, nombre, nit,</code> <code>... }</code>

<code>actualizarCliente()</code>	<code>ActualizarClienteRequest { id, nombre?, ... }</code>	<code>ClienteResponse</code> (datos actualizados)
<code>eliminarCliente()</code>	<code>clienteId: UUID</code>	<code>void</code> (o un estatus)

Por ejemplo, para crear un cliente se esperaría una petición REST como:

```

1 POST /api/clientes
2 {
3   "nombre": "Empresa XYZ",
4   "tipoIdentificacion": "NIT",
5   "numeroDocumento": "1020304050",
6   "correo": "contacto@xyz.com",
7   "telefono": "5551234",
8   "direccion": "Av. Principal 123"
9 }
```

El servicio `GestionClientesService.createCliente` procesaría esto y devolvería (HTTP 201) una respuesta JSON con el cliente creado:

```

1 {
2   "id": "a1b2c3d4-e5f6-7890-1234-56789abcdef0",
3   "nombre": "Empresa XYZ",
4   "tipoIdentificacion": "NIT",
5   "numeroDocumento": "1020304050",
6   "correo": "contacto@xyz.com",
7   "telefono": "5551234",
8   "direccion": "Av. Principal 123"
9 }
```

Internamente, `createCliente()` podría hacer algo como `ClienteFactory.create(request)` para crear el agregado, usar `ClienteRepository.save(cliente)`, y luego publicar el evento `ClienteRegistrado`. Toda esta coordinación de repositorio y publicación de eventos se realiza en el Application Service.

# Infraestructura y Adaptadores\_gc

## 8.1 Repositorios

En la capa de infraestructura se implementan los repositorios para persistir el modelo de dominio. Por ejemplo, usando Spring Data JPA podríamos tener:

```
1 public interface ClienteRepository extends JpaRepository<ClienteEntity, UUID> {  
2     // Consultas adicionales, p.ej. para verificar duplicados:  
3     Optional<ClienteEntity> findByNit(String nit);  
4 }
```

El repositorio actúa como mediador entre el modelo de dominio (`Cliente`) y la base de datos. Oculta detalles de SQL u ORM, exponiendo métodos centrados en el dominio (p.ej. `save()`, `findById()`, consultas específicas). Así se preserva la pureza del dominio y la lógica de persistencia queda aislada.

**Multitenancy:** Como Factura Fácil es multitenant, se selecciona una estrategia adecuada (por ejemplo, base de datos compartida con columna `tenant_id`, esquema por empresa o base de datos por empresa). Spring Data JPA o Hibernate pueden configurarse para filtrar automáticamente por el tenant actual (obtenido del contexto de seguridad). La infraestructura de repositorio incluye esta lógica de multitenancy para asegurar que cada empresa vea solo sus clientes.

## 8.2 Gateways/Adaptadores de Mensajería

Para integración asíncrona se definen adaptadores a sistemas de mensajería. Por ejemplo, un **KafkaEventGateway** o **RabbitMqAdapter** que publique eventos de integración en tópicos/queues. El patrón de Mensajería permite desacoplar contextos. Apache Kafka suele usarse para arquitecturas basadas en eventos de dominio, ya que combina publicación/suscripción, almacenamiento persistente y escalabilidad. RabbitMQ se podría usar para colas de tareas o routing complejo.

- **Kafka:** se configuran producers que publican eventos (por ejemplo, el evento `ClienteRegistrado`) en un topic “clientes”. Otros microservicios (como ComprobantesElectronicosBC) podrán suscribirse a ese topic. Kafka mantiene un log duradero de eventos y permite a nuevos consumidores leer desde el pasado. Como apuntan Dev Cookies, “Kafka sobresale en arquitecturas event-driven con alto *throughput*, persistencia y escalabilidad”.
- **RabbitMQ:** útil para patrones de cola de trabajo o mensajería confiable. Un uso podría ser enviar tareas (p.ej. notificaciones o integraciones no críticas) en colas con ack. RabbitMQ ofrece mecanismos de reintento y enrutamiento (exchanges/filtros) que pueden usarse según necesidad.

En ambos casos, se implementan adaptadores (por ejemplo, con Spring Cloud Stream) que traducen los eventos de dominio (`ClienteRegistrado`) a mensajes del bus. Estos adaptadores se inyectan en los Application Services o Domain Services apropiados. La infraestructura configura temas/colas y gestión de conexión. Tal como sugiere la documentación, “la interfaz de bus de eventos necesita una infraestructura que permita comunicación entre procesos... [p.ej.] colas o un bus de servicio”.

## 8.3 API (REST/GraphQL/WS)

El servicio GestiónClientesBC se expone típicamente vía API REST (p.ej. endpoints HTTP bajo `/api/clientes`). Se podría usar **Spring MVC** o **Spring WebFlux** para definir controladores que llamen a los Application Services.

Alternativamente, se puede ofrecer un endpoint GraphQL para consultas flexibles. Para notificaciones en tiempo real (p.ej.

actualización de UI de cliente), se podrían usar WebSockets o SSE, pero esto depende de los requerimientos del cliente. En cualquier caso, la capa de presentación se conecta a la aplicación a través de interfaces definidas.

#### 8.4 EventBusAdapter

Además de Kafka/RabbitMQ, se puede incluir un **EventBusAdapter** interno (por ejemplo, con Guava EventBus o un bus propio) para manejar eventos de dominio dentro del mismo servicio (sin salir de la JVM). Los *Domain Events* generados en las entidades se envían a este bus interno, donde handlers locales los capturan para ejecutar lógica adicional (coherencia interna o publicación a la capa de integración). En cambio, los *Integration Events* se envían al bus externo (Kafka, Rabbit). Como indica Microsoft, los eventos de dominio se mantienen dentro del bounded context y pueden ser sincrónicos, mientras que los eventos de integración se basan en mensajería asincrónica hacia otros contextos.

#### 8.5 Patrones de Infraestructura

En la implementación se usan varios patrones de diseño comunes:

- **Factory:** para creación de agregados completos. Por ejemplo, `ClienteFactory.create(request)` se encarga de instanciar `Cliente` (o `ClienteEntity`) con sus invariantes iniciales.
- **Handler (Command/Event Handler):** clases que manejan comandos o eventos recibidos (p.ej. `RegistrarClienteHandler`, `ClienteRegistradoEventHandler`). A menudo se integran con frameworks como Spring para invocar estos métodos al recibir mensajes.
- **Mapper:** para convertir entre entidades JPA y objetos del dominio o DTOs. Por ejemplo, se usa un mapper (manual o librerías como MapStruct) para traducir `ClienteEntity` ↔ `Cliente` (VOs) y a los DTOs de API. Esto aísla las capas y simplifica la transformación de datos.

Estos patrones ayudan a estructurar el código y separar responsabilidades, tal como se recomienda en DDD. Por ejemplo, el patrón Repository (mencionado antes) es clave para separar la persistencia del dominio

## Mapa de Contextos\_gc



- **ComprobantesElectonicosBC:** Este contexto consume datos de cliente al generar facturas. GestiónClientesBC actúa como *Supplier* y ComprobantesElectonicosBC como *Customer*. Hay un contrato (API o eventos) para compartir datos de cliente. Si los modelos difieren, se puede usar una capa Anti-Corrupción (ACL) para traducir (p.ej. adaptar el objeto `Cliente` al modelo de facturación). Esta relación se etiqueta típicamente como *Customer/Supplier*.
- **IndicadoresNegocioBC:** Es un contexto de análisis que suscribe eventos para generar métricas (clientes nuevos, facturación por cliente, etc.). Recibe los eventos publicados por GC (ClienteRegistrado, ClienteActualizado) y por CE (FacturaCreada) para actualizar indicadores. La comunicación es unidireccional (GC/CE → IND) mediante eventos. Se podría considerar un estilo *Customer/Supplier* donde Indicadores es consumidor downstream.
- **ControlCajaBC, CatalogoArticulosBC, ListaPreciosBC, ConfiguracionSistemaBC:** Estos contextos no requieren datos de clientes de GestiónClientesBC en tiempo real. Por lo tanto, se consideran casos de "**Separate Ways**": no hay integración técnica con GestiónClientesBC, sino que operan de forma independiente. Por ejemplo, el manejo de caja (ventas) o catálogos no necesitan llamar a operaciones de cliente en este contexto. Se documenta esta decisión para evitar integraciones innecesarias.
- **Mi Contasis (Identidad/Autenticación):** Es un sistema externo que provee usuarios y tenancy. GestiónClientesBC delega autenticación a Mi Contasis vía el *Orquestador de Apps*. Esencialmente, Mi Contasis es upstream que valida tokens y resuelve la empresa (tenant) antes de permitir operaciones en GestiónClientes. En términos de DDD, es una relación *Customer/Supplier* (o *Conformist*), donde GestiónClientesBC consume el servicio de identidad de Mi Contasis sin adaptar su modelo de dominio. Todo servicio verifica las credenciales con Mi Contasis para obtener el ID de tenant antes de acceder a los datos.

En el **Context Map** global se reflejan estos vínculos: los equipos acuerdan contratos (APIs o eventos) y patrones de integración (p.ej. ACL o Conformist) según la necesidad. Los contextos sin integración se marcan con la política *Caminos Separados*. Este mapa ayuda a visualizar dependencias y garantizar que la interacción entre dominios siga el lenguaje compartido.

# Política de Consistencia y Transaccionalidad\_gc

## 10.1 Consistencia eventual y Sagas ☺

Para mantener escalabilidad y desacoplo entre contextos, se adopta **consistencia eventual** en lugar de transacciones distribuidas. Cada contexto maneja su propia transacción local y publica eventos que otros contextos consumen de forma asíncrona. Las operaciones que abarcan múltiples BC se implementan como **sagas**. Según el patrón Saga, “se implementa cada transacción de negocio como una secuencia de transacciones locales. Cada transacción local actualiza su base de datos y publica un mensaje/evento para desencadenar la siguiente”. Por ejemplo, al registrar un cliente en GestiónClientesBC se publica `ClienteRegistrado`; luego, otro servicio (p.ej. de facturación o indicadores) reaccionaría a este evento para completar su parte del flujo.

La coordinación de la saga puede ser **coreográfica** o **orquestada**. En coreografía, cada servicio participa publicando eventos que activan a los demás, sin un controlador central. En orquestación, un componente central dirige los pasos enviando comandos específicos. En este diseño suele preferirse la *coreografía*, minimizando acoplamientos explícitos: cada bounded context reacciona a los eventos según convenga. Si se requiere control más fino, se puede incorporar un **orquestador de sagas** que se comunique con los participantes mediante mensajes.

## 10.2 Orquestador y transacciones compensatorias ☺

En la implementación de sagas, un **orquestador de transacciones** puede gestionar el flujo principal. Este componente (o patrón) envía comandos a cada servicio participante y espera sus respuestas (o consume sus eventos) para decidir el siguiente paso. Si alguna transacción local falla, el orquestador debe disparar **transacciones compensatorias** para deshacer los pasos previos. Por ejemplo, si luego de crear un cliente en GC algo falla en Facturación, se puede definir un comando compensatorio (p.ej. `EliminarCliente`) para revertir la operación.

En resumen, se evita la consistencia fuerte (2PC) en favor de **coherencia eventual**: los BC se actualizan de forma asíncrona, sincronizando su estado mediante eventos. Las sagas garantizan que, aun en caso de errores, el sistema llegue a un estado consistente parcial usando compensaciones. Esta estrategia permite alto rendimiento y tolerancia a fallos, delegando a los expertos de dominio la decisión de qué operaciones requieren estricta atomicidad y cuáles pueden ser eventualmente consistentes.

**Fuentes:** Los conceptos de sagas y patrones de integración provienen de la literatura sobre microservicios y DDD, mientras que las decisiones de arquitectura multitenant y repositorios se basan en prácticas recomendadas (e.g. multitenancy con JPA y el patrón Repository como se detalla en DDD).

## Estrategia de Persistencia y Esquema de Datos\_gc

El contexto de **Gestión de Clientes** almacena información de clientes asociados a cada empresa (tenant). La persistencia debe soportar multi-tenant, por lo que cada registro incluye el identificador de la empresa. Se sugiere una tabla (o colección) principal llamada **Cliente**, con campos como se detalla a continuación. Cada cliente se identifica de forma única (UUID) y está asociado a un `empresaId` que denota al tenant. Otros campos comunes son nombre, datos de contacto y metadatos de auditoría.

Campo	Tipo de dato	Descripción
<code>idCliente</code>	UUID (String)	Identificador único del cliente (PK).
<code>empresaId</code>	UUID (String)	Identificador de la empresa/tenant.
<code>nombre</code>	String	Nombre o razón social del cliente.
<code>tipoDocumento</code>	String	Tipo de documento (por ejemplo, "DNI" o "RUC").
<code>numeroDoc</code>	String	Número del documento de identidad o registro legal.
<code>correo</code>	String	Correo electrónico del cliente.
<code>telefono</code>	String	Teléfono de contacto.
<code>fechaRegistro</code>	Timestamp	Fecha y hora en que se creó el registro.

```
1 {
2   "idCliente": "123e4567-e89b-12d3-a456-426655440000",
3   "empresaId": "abc123de-45f6-7890-gh12-ijkl345mno67",
4   "nombre": "ACME S.A.",
5   "tipoDocumento": "RUC",
6   "numeroDoc": "20123456789",
7   "correo": "ventas@acme.com",
8   "telefono": "+51987654321",
9   "fechaRegistro": "2025-07-01T09:30:00Z"
10 }
```



## 11.2 Índices y particionamiento

Para optimizar consultas multi-tenant, se recomienda un índice por `empresaId` y, si hay búsquedas frecuentes por documento, un índice compuesto único (`empresaId, numeroDoc`). El `idCliente` es la clave primaria. Otros índices útiles podrían ser en el campo `nombre` para búsquedas por texto. En bases relacionales, se podría particionar la tabla por rango de `empresaId` o usar esquemas separados por tenant para escalar horizontalmente. En bases NoSQL, se puede usar una colección por empresa o un campo `empresaId` como key de partición.

- **Índices sugeridos:** PK sobre `idCliente`; índice filtrante sobre `empresaId`; índice único (`empresaId, numeroDoc`); índice full-text en `nombre` si se requiere búsqueda avanzada.
- **Particionamiento:** usar particiones de tabla por rango de `empresaId` en SQL, o separar esquemas/bases por empresa; en NoSQL emplear shard key en `empresaId`. Esto aisla datos por tenant y mejora el rendimiento global.

## 11.3 Versionado y migraciones

Las migraciones de esquema deben versionarse rigurosamente. Se recomienda usar herramientas como Flyway o Liquibase dentro del CI/CD para aplicar cambios controlados. Cada migración debe llevar un identificador de versión, de modo que en despliegues multi-tenant se pueda aplicar en orden. Para multiempresa, las migraciones pueden ejecutarse en una base compartida (paso único) o iterar por cada base de datos por tenant. Es clave mantener scripts inmutables y documentar cada cambio. En caso de tablas eventuales (como logs de auditoría), seguir tablas append-only con versión. Además, mantener una carpeta de versiones en el repositorio del BC permite **rollback** controlado si hay fallos.

# Seguridad y Autorización\_gc

## 12.1 Autenticación centralizada

La autenticación no la maneja directamente este BC. El dominio de **Identidad y Autenticación (Mi Contasis)** es responsable de emitir tokens JWT (OAuth2) que incluyen información del usuario, roles y tenant. El *Orquestador de Apps* administra los tenants, asignando permisos iniciales y configuraciones por empresa. Al recibirse una petición al BC de GestiónClientes, el **Gateway/API** verifica el JWT contra Mi Contasis. Así, el BC se enfoca en la lógica de clientes; la autenticación y la gestión de tenants quedan delegadas a los dominios correspondientes.



## 12.2 Roles y permisos

Se definen roles específicos para las operaciones del módulo de clientes. Por ejemplo:

- **Rol\_AdminCliente:** administrador completo del módulo. Permite crear, leer, actualizar y eliminar clientes para la empresa asignada.
- **Rol\_Asistente:** usuario con permisos limitados. Puede crear y consultar clientes, pero típicamente no puede eliminarlos.
- **Rol\_ConsultorClientes:** rol de solo lectura. Permite ver información de clientes y reportes, sin modificaciones.

Estos roles se incluyen en los tokens JWT emitidos. Al procesar cada petición, el BC valida que el usuario tenga el rol adecuado para la operación solicitada.

Rol	Descripción	Permisos CRUD
Rol_AdminCliente	Administra todos los clientes	Create, Read, Update, Delete
Rol_Asistente_Sis	Asistente del sistema (limitado)	Create, Read, (sin Delete)
Rol_Consultor	Consultas y reportes (solo lectura)	Read

## 12.3 Contratos y validaciones

Para garantizar la autorización, el API Gateway ejerce como punto de entrada seguro: verifica el JWT (firma, expiración, roles) antes de enrutar al servicio. En los microservicios del BC se aplican anotaciones de seguridad (por ejemplo, en Spring Security usar `@PreAuthorize("hasRole('Rol_AdminCliente')")`) para doble validación. Los mensajes/eventos que se intercambian con otros BCs incluyen campos de contexto (ej. `empresaId`, `usuarioId`) para que los servicios receptores validen permisos. Por ejemplo, al enviar un evento `ClienteCreado`, se incluye el usuario que lo creó y su tenant. De esta manera, todos los niveles (gateway, controlador, lógica de dominio) aplican validaciones de roles según los contratos definidos en la documentación interna del sistema.

# Monitoreo, Métricas y Alertas\_gc

## 13.1 Métricas clave ↗

Se debe recolectar métricas tanto de negocio como técnicas para este BC. Algunas métricas recomendadas son:

- **Clientes nuevos:** Número de clientes dados de alta por día o periodo. Indicador clave de adopción.
- **Tasa de errores en validaciones:** Proporción de peticiones rechazadas por datos inválidos. Ayuda a detectar cambios de comportamiento.
- **Latencia de operaciones:** Tiempo de respuesta para búsquedas, creación o actualización de clientes (p95, p99).
- **Errores del servicio:** Tasa de errores internos (códigos 5xx). Monitorea la salud del servicio.
- **Uso de recursos:** CPU, memoria y consumo de GC de la instancia. Detecta cuellos de botella.
- **Eventos pendientes:** Tamaño de colas o topics (Kafka) relacionados con clientes, para monitorear cuellos de integración.

## 13.2 Herramientas e integración ↗

Se sugiere instrumentar el servicio con **Prometheus** para recolección de métricas (usando, por ejemplo, Spring Actuator/Micrometer). Los indicadores expuestos (/metrics) son recogidos por Prometheus. Para visualización y dashboards, se usa **Grafana**, creando paneles con gráficos históricos. Cada métrica mencionada se mapea a un panel: cantidad de clientes, tasa de errores, latencias (gráfico de histograma). El sistema de alertas de Prometheus (Alertmanager) se integra para notificaciones (email, Slack, etc.). En resumen:

- *Prometheus*: scraping de métricas de la API (`/actuator/prometheus`).
- *Grafana*: dashboards de negocio y técnicos.
- *Alertmanager*: envía alertas cuando se superan umbrales definidos.

## 13.3 Alertas, SLIs y SLAs ↗

Se definen SLIs (Service Level Indicators) y alertas basadas en ellos. Por ejemplo, un SLI clave es la **disponibilidad** del servicio (porcentaje de peticiones exitosas), y otro es la **latencia p95** de respuestas. Un SLA interno podría ser: “el servicio de clientes debe responder al 99.9% de peticiones en menos de 200ms”. Las alertas se configuran cuando se rompen estos umbrales: p. ej., si el error rate supera el 5% en 5 minutos o la latencia p95 excede 300ms en 3 minutos, se activa una alerta. Otros SLI típicos son la **tasa de error** y la **saturación de recursos**. En resumen:

- **SLIs propuestos:** Disponibilidad (uptime), error rate, latencia (p95/p99), throughput de peticiones.
- **Alertas configuradas:** Error rate >5% sostenido, latencia > umbral, CPU >80%, disk casi lleno, etc.
- **SLA ejemplar:** 99.9% de peticiones exitosas con latencia < 200ms durante horas laborales.

# Estrategia de Testing\_gc

## 14.1 Pruebas unitarias

Se implementan pruebas de unidad para componentes del dominio (agregados, servicios de dominio, validadores). Cada caso de uso principal (crear cliente, actualizar datos, validar duplicados) tiene tests con JUnit. Se emplean mocks (por ejemplo Mockito) para simular repositorios o servicios externos. Las pruebas validan la lógica de negocio: cálculo de restricciones, reglas de validación de documentos, etc.

## 14.2 Pruebas de contrato e integración

Para asegurar la comunicación con otros BCs y mensajería, se usan pruebas de contrato. Por ejemplo, al consumir el dominio de Identidad, se verifica que la respuesta del servicio de autenticación cumple el esquema esperado (claims en JWT). Para eventos asíncronos, se emplea **Pact** o **Spring Cloud Contract**: se prueba que el evento `ClienteCreado` emitido cumple el formato pactado y que el consumidor puede procesarlo. Estas pruebas garantizan que los contratos de mensajes no se rompan entre despliegues.

## 14.3 Pruebas end-to-end (E2E)

Se realizan pruebas de flujo completo de gestión de clientes. Por ejemplo, un escenario E2E: *un usuario con Rol\_AdminCliente crea un cliente nuevo vía API, se guarda en la BD, se verifica audit trail y luego se consulta este cliente para confirmar datos*. Estas pruebas pueden automatizarse con Postman, RestAssured o marcos BDD (Cucumber). El objetivo es validar el sistema en conjunto, incluyendo base de datos real (o Docker), asegurando que todos los componentes integrados funcionan como se espera desde la entrada hasta la persistencia.

## 14.4 Frameworks sugeridos

Se recomienda usar herramientas de pruebas modernas en Java/Spring:

- **JUnit 5**: pruebas unitarias de servicios, validadores y repositorios.
- **Spring Test (Spring Boot Test)**: pruebas de integración que cargan contexto Spring (mockMVC, pruebas de endpoints).
- **Mockito**: creación de mocks para dependencias internas en pruebas unitarias.
- **TestContainers**: despliegue de contenedores efímeros (PostgreSQL, Kafka) durante pruebas de integración.
- **Pact (o Spring Cloud Contract)**: pruebas de contrato entre servicios (mensajes/events).

Cada prueba debe correr en el pipeline CI para asegurar calidad continua.

## Decisiones de Arquitectura (ADR)\_gc

- **Event Bus vs REST síncrono:** Se opta por un *bus de eventos* (Kafka) para la mayoría de integraciones, desacoplando los microservicios. Esto permite escalabilidad y resiliencia. Las APIs REST síncronas se usan solo para consultas puntuales internas; para integración entre dominios predomina la comunicación basada en eventos.
- **Relacional vs NoSQL:** Se elige base de datos relacional (por ejemplo PostgreSQL) para mantener integridad referencial y transaccional de datos de clientes. Las relaciones (empresa – cliente) y las reglas de negocio complejas se modelan mejor en SQL. NoSQL solo se consideraría si hubiera requerimientos de alta escritura horizontal o datos no estructurados (no es el caso).
- **Patrones de integración:** Se aplica el patrón *Event-driven* (publicar/subscribir eventos de dominio). También se usan *ACL (Anti-Corruption Layer)* al integrarse con sistemas externos para traducir modelos. Para servicios internos, se sigue CQRS simplificado: comandos sincronizados via REST cuando es necesario, y eventos dominan el flujo.
- **Herramientas de mensajería y orquestación:** Se selecciona **Kafka** como backbone de mensajería por su alto rendimiento y durabilidad. Para colas con requerimientos específicos (ej. procesamiento asíncrono puntual), se podría utilizar **RabbitMQ**. El orquestador de aplicaciones administra la creación y configuración de temas/colas por empresa. Todas las decisiones quedan documentadas como parte de los ADR internos.



Cada decisión clave se registra en un ADR (Architecture Decision Record), facilitando la trazabilidad de los criterios técnicos adoptados en el proyecto.

# ListaPreciosBC - Antonio H

El Bounded Context **ListaPreciosBC** gestiona la definición y aplicación de las listas de precios en Factura Fácil, un sistema multitenant orientado a microempresas. Este BC centraliza la lógica de tarifas, de modo que al emitir comprobantes el sistema seleccione automáticamente el precio correcto según criterios configurables (cliente, canal, volumen, promociones). A su vez, mantiene un historial completo de cada cambio de precio para auditoría y revertir ajustes pasados. ListaPreciosBC se integra con otros contextos (por ejemplo, consume eventos de Gestión de Clientes o Catálogo de Artículos) para actualizar sus reglas de precio, y publica eventos (p.ej. precio aplicado) hacia indicadores de negocio y otros módulos.

## 1.1 Propósito

El propósito de **ListaPreciosBC** es **centralizar la lógica de definición, aplicación y auditoría de tarifas** dentro de Factura Fácil. Esto implica gestionar listas de precios diferenciadas por criterios de negocio (segmento de cliente, canal de venta, volúmenes, promociones, etc.) y garantizar que, al emitir un comprobante electrónico, se seleccione automáticamente el precio correcto. De esta forma, el sistema evita descuentos manuales y errores de facturación, y además registra la trazabilidad completa de cada ajuste tarifario. En resumen, ListaPreciosBC asegura la consistencia y flexibilidad en la política de precios, cubriendo las necesidades de PYMEs con catálogos crecientes y campañas dinámicas.

## 1.2 Responsabilidades Clave

ListaPreciosBC se encarga principalmente de **gestionar y aplicar reglas de precios**. Entre sus responsabilidades destacan:

- **Configuración de listas de precios:** Permite crear, editar y desactivar múltiples listas de precio por producto/servicio (hasta 10 por ítem), cada una con criterios específicos (por cliente, canal, volumen o promoción). En cada lista se definen atributos como tipo de lista (CLIENTE, CANAL, VOLUMEN, PROMOCIÓN), moneda y rango de aplicación.
- **Definición de atributos tributarios y contables:** Para cada lista o precio específico, se establece cómo se grava el ítem (IGV gravado, exonerado, inafecto, detacciones, etc.) y se definen conversiones de unidades y cuentas contables asociadas.
- **Aplicación automática de precios:** Al generarse un comprobante, el BC evalúa las listas activas en orden de prioridad (Cliente > Canal > Volumen > Promoción > Base) y selecciona el **PrecioEspecifico** correspondiente. Este cálculo se puede encapsular en un servicio de dominio especializado en determinar el precio a aplicar.
- **Gestión de vigencias y expiraciones:** Controla las fechas de inicio/fin de cada lista y las condiciones de volumen. Activa o marca como expirada una lista cuando excede su periodo o volumen asignado.
- **Histórico inmutable de cambios:** Registra **en forma inmutable** todo ajuste tarifario. Cada vez que se agrega, modifica o elimina un **PrecioEspecifico**, se genera un registro en el **HistorialPrecio** con usuario, fecha y motivo. Esto asegura cumplimiento de auditorías y permite analizar la evolución de precios.
- **Operaciones masivas:** Soporta importación y exportación de listas de precios vía archivos (p.ej. Excel) para crear o actualizar en lote múltiples listas.
- **Integración con otros BC y notificaciones:** Publica eventos relevantes hacia otros contextos. Por ejemplo, notifica al equipo comercial al cambiar listas clave o expirar promociones. Además, transmite el evento

PrecioAplicado a IndicadoresNegocioBC para alimentar métricas de margen y variación, y podría sincronizar segmentos con GestiónClientesBC (evento *ListaPrecioClienteModificada*).

## 1.3 Contexto y Motivación

En el contexto de microempresas (PYMEs) multitenant que usan Factura Fácil, **los precios suelen variar según reglas de negocio complejas**. Factores como el tipo de cliente, el canal de venta, la cantidad comprada o las promociones temporales pueden afectar las tarifas. Por ello, se requiere flexibilidad tarifaria sin introducir carga operacional. ListaPreciosBC nace para resolver estos desafíos:

- **Evitar errores humanos:** Al centralizar la lógica de precios, se automatizan descuentos y promociones. Esto reduce errores en la facturación y evita omisiones en las tarifas especiales.
- **Auditoría y cumplimiento:** Mantener un historial completo de cambios de precio protege contra fraudes internos y cumple con normativas internas de control. Cada cambio queda registrado con su contexto (usuario, fecha, motivo).
- **Escalabilidad operativa:** A medida que crece el catálogo de productos y las campañas promocionales son más dinámicas, ListaPreciosBC permite gestionar masivamente múltiples listas y criterios en paralelo. La arquitectura basada en microservicios y eventos facilita escalar este BC según demanda.
- **Mejorar insights de negocio:** Al centralizar datos de precios y cambios, se potencia el análisis de márgenes y la eficacia de las promociones a través de indicadores, aportando valor estratégico (ver secc. 1.8).

## 1.4 Fuentes de Información

El BC provee y consume información clave, estructurada en casos de uso principales:

- **Configurar listas de precios:** Operación de creación/edición de listas. Permite al usuario definir una lista nueva o actualizar una existente con criterios de cliente, canal, volumen o campaña.
- **Asignar precio automáticamente:** Cuando se emite un comprobante, el BC determina qué **PrecioEspecifico** aplicar. Este proceso sigue la prioridad establecida: Cliente > Canal > Volumen > Promoción (según RN-LP-001).
- **Gestionar vigencias y expiraciones:** El sistema chequea periódicamente o al editar la lista las fechas de inicio y fin, así como tramos de volumen. Una vez que una lista alcanza su fin de vigencia o agota su rango de volumen, se marca como expirada.
- **Importación/Exportación masiva:** Se procesan archivos Excel u otros formatos para crear o actualizar múltiples listas de precios en lote. Esto facilita la carga inicial o ajustes masivos sin intervención manual ítem por ítem.
- **Registrar historial de cambios:** Cada vez que se modifica un precio, se genera automáticamente un registro en el historial. Este registro incluye metadatos (usuario, fecha, motivo). La información histórica permanece inmutable.
- **Notificar al equipo comercial:** Al cambiar listas clave (p.ej. en segmentos importantes) o al expirar promociones, el BC envía alertas a través de notificaciones o eventos para que el área comercial esté informada.

Cada funcionalidad utiliza datos de otros contextos (por ejemplo, información de cliente o producto) y expone servicios/eventos para el resto del sistema.

## 1.5 Modelo de Dominio

**ListaPreciosBC** se estructura en torno al agregado raíz **ListaPrecio**, que agrupa uno o más precios específicos bajo un criterio común. A continuación se describen los elementos principales del modelo:

- **Agregado raíz:** **ListaPrecio**. Representa una colección de entradas tarifarias (**PrecioEspecifico**) asociadas a un mismo criterio (por ejemplo, “clientes VIP” o “canal online”). Entre sus operaciones clave están *agregarPrecioEspecifico()*, *actualizarPrecio()* e *inhabilitarLista()*.

- **Entidad:** `PrecioEspecifico`. Define un precio único para un criterio dado. Contiene atributos como el valor numérico, la moneda, la prioridad de aplicación (menor = mayor prioridad) y la vigencia (fechas o volumen). Incluye comportamientos para validar el valor, verificar vigencia y aplicar su prioridad dentro de la lista. Cada `PrecioEspecifico` pertenece a una sola `ListaPrecio`.
- **Entidad:** `HistorialPrecio`. Registro inmutable que almacena cada modificación de un `PrecioEspecifico`. Contiene datos de auditoría (usuario que hizo el cambio, fecha, motivo). Está relacionado con el `PrecioEspecifico` modificado (N:1).
- **Entidad/VO:** `CriterioLista`. Especifica el criterio de la lista: puede incluir `clienteId` (para listas por cliente), `canalVenta` (p.ej. tienda física o online), `rangoVolumen` o `periodoVigencia`. Aunque inicialmente se concibe como un objeto de valor, aquí funciona conceptualmente como parte del estado del agregado, determinando cuándo aplicar la lista.
- **Objetos de Valor:**
  - `TipoLista` (enum): clasifica el tipo de lista (CLIENTE, CANAL, VOLUMEN, PROMOCIÓN). Define el orden de prioridad de la lista y su lógica de aplicación.
  - `Moneda` (enum): indica la moneda del precio (PEN, USD). Garantiza consistencia en los cálculos.
  - `RangoVolumen` (VO): estructura inmutable con `{cantidadMin, cantidadMax}`. Sirve para definir descuentos por volumen; los rangos no deben solaparse (`cantidadMin < cantidadMax`).
  - `PeriodoVigencia` (VO): estructura inmutable con `{fechaInicio, fechaFin}`. Se usa para promociones temporales; debe ser coherente (`fechaInicio < fechaFin`).
- **Servicios de Dominio:** Aunque la mayor parte de la lógica reside en entidades, pueden definirse servicios específicos para operaciones transversales. Por ejemplo, un **Servicio de Selección de Precio** que, dado un contexto de venta, recorra las listas activas en orden de prioridad (según RN-LP-001) para devolver el `PrecioEspecifico` aplicable. Otro podría ser un **Servicio de Validación de Listas**, encargado de comprobar restricciones complejas (unicidad de listas por criterio, vigencia, tramos no solapados) antes de persistir cambios.

La siguiente tabla resume el modelo de dominio (agregado, entidades y VO):

Elemento	Tipo	Descripción Breve
<code>ListaPrecio</code>	Agregado Raíz	Agrupa varios <code>PrecioEspecifico</code> bajo un criterio (cliente, canal, etc.). Métodos: <code>agregarPrecioEspecifico()</code> , <code>actualizarPrecio()</code> , <code>inhabilitarLista()</code> .
<code>PrecioEspecifico</code>	Entidad	Define el precio (valor, moneda), prioridad y vigencia para un criterio. Se valida su valor (>0) y vigencia antes de usarlo. Relación 1:N con <code>ListaPrecio</code> .
<code>HistorialPrecio</code>	Entidad	Registro inmutable de cada cambio en un <code>PrecioEspecifico</code> . Contiene usuario, fecha y motivo.

		Relación N:1 con <b>PrecioEspecifico</b> .
<b>CriterioLista</b>	(VO/Entidad)	Especifica el criterio de aplicación (ID de cliente, canal, <b>RangoVolumen</b> , <b>PeriodoVigencia</b> ). Asocia condiciones a la lista.
<b>TipoLista</b>	VO (enum)	{CLIENTE, CANAL, VOLUMEN, PROMOCIÓN}. Indica el tipo de la lista y establece prioridad relativa.
<b>Moneda</b>	VO (enum)	{PEN, USD}. Moneda de los precios.
<b>RangoVolumen</b>	VO	{cantidadMin, cantidadMax} inmutables. Uso: descuentos escalonados por volumen.
<b>PeriodoVigencia</b>	VO	{fechaInicio, fechaFin} inmutables. Uso: promociones temporales.

## 1.6 Relaciones con otros BC ↗

ListaPreciosBC colabora con los siguientes contextos existentes:

- **ComprobantesElectronicosBC (Inbound):** Consume el evento **ComprobanteEmitido** (que incluye clientId, producto/servicioId, cantidad, fecha) para calcular el precio correspondiente al generar la factura o boleta.
- **CatalogoArticulosBC (Inbound):** Atiende los eventos **ProductoModificado** o **ProductoInhabilitado**. Si cambian atributos de un ítem (p.ej. unidad de medida, afectación de IGV, etc.), puede ser necesario actualizar o invalidar listas de precios asociadas.
- **GestionClientesBC (Inbound):** Atiende el evento **ClienteModificado**. Cuando cambia el segmento o estado de un cliente, se actualizan las listas de precios específicas por cliente.
- **ConfiguracionSistemaBC (Inbound):** Recibe actualizaciones en parámetros globales (**ParametrosPreciosActualizados**) como moneda por defecto, política de redondeo o límites de tramos. Estos afectan la validación y cálculos de las listas de precios.
- **IndicadoresNegociosBC (Outbound):** Publica el evento **PrecioAplicado** tras cada emisión de comprobante con el precio finalmente aplicado. Este evento es consumido por IndicadoresNegociosBC para actualizar dashboards de márgenes y seguimiento de promociones.
- **GestionClientesBC (Outbound):** Publica el evento **ListaPrecioClienteModificada** cuando se altera la relación de listas con clientes. Esto permite a Gestión de Clientes sincronizar la segmentación de clientes con precios especiales.

**Nota:** No se definen BCs adicionales (“ProductosBC”, “VentasBC”, etc.), conforme al dominio oficial. Todas las referencias a productos o ventas se manejan a través de los contextos existentes (Catálogo de Artículos, Comprobantes Electrónicos, etc.).

### 1.6.1 Consumo de Eventos (Inbound)

Los eventos **entrantes** que este contexto **consume** de otros bounded contexts le permiten reaccionar ante cambios externos y mantener la coherencia de los precios:

Evento Consumido (Fuente)	Datos / Propósito
<b>ComprobanteEmitido</b> <i>(ComprobantesElectronicosBC)</i>	Recibe los datos de un comprobante emitido (cliente, productos/servicios, cantidad, fecha, etc.) para calcular y aplicar el precio correspondiente durante la facturación. Este evento asegura que al generarse una venta, se determine automáticamente el <b>precio aplicado</b> según las listas vigentes.
<b>ProductoModificado /</b> <b>ProductoInhabilitado</b> <i>(CatalogoArticulosBC)</i>	Cuando un producto es actualizado (por ejemplo, cambio de impuesto, unidad de medida) o deshabilitado en el catálogo, ListaPreciosBC verifica si alguno de los precios especiales asociados queda inválido o requiere ajuste. De este modo se mantiene la consistencia entre la información de producto y las tarifas definidas.
<b>ClienteModificado</b> ( <i>GestionClientesBC</i> )	Al modificarse datos de un cliente (p. ej. cambio de segmento de cliente o estado activo/inactivo), este evento permite actualizar o reevaluar listas de precios especiales ligadas a ese cliente. Por ejemplo, si un cliente VIP cambia de categoría, podrían activarse o desactivarse ciertos precios preferenciales.
<b>ParametrosPreciosActualizados</b> <i>(ConfiguracionSistemaBC)</i>	Notifica cambios en parámetros globales de precios, como la moneda por defecto del sistema, políticas de redondeo o límites de rangos de volumen. ListaPreciosBC ajusta su comportamiento (cálculos, validaciones) en base a estos

parámetros para seguir las políticas vigentes definidas a nivel central.



## 1.8 Valor Estratégico ↗

ListaPreciosBC aporta un valor significativo al negocio de Factura Fácil al:

- **Maximizar ingresos:** Asegura que cada segmento de cliente y volumen aplique la tarifa óptima, evitando descuentos inadvertidos.
- **Reducir errores y disputas:** Al automatizar la aplicación de precios correctos, minimiza reclamos o correcciones posteriores por tarifas incorrectas.
- **Acelerar campañas:** Facilita la configuración rápida de promociones y cambios masivos de precios, agilizando la respuesta ante ofertas o cambios de mercado.
- **Mejorar análisis:** Proporciona datos consistentes para el cálculo de márgenes y eficacia de promociones, alimentando los indicadores de negocio con información confiable.
- **Cumplir auditorías:** Mantener el historial completo de cambios de precio permite un seguimiento detallado en auditorías internas o externas, aumentando la confianza de usuarios y reguladores.

En conjunto, la gestión de listas de precios centralizada incrementa la competitividad y la eficiencia operativa de microempresas multitenant en el ecosistema Factura Fácil.

**Fuentes:** Esta documentación se alinea con el modelo de dominio oficial de Factura Fácil, sin introducir nuevos contextos que no existan en el dominio establecido.1. Propósito

ListaPreciosBC centraliza la lógica de definición, aplicación y gestión histórica de tarifas en **Factura Fácil**. Garantiza que, al momento de emitir comprobantes, el sistema seleccione automáticamente el precio correcto según criterios del cliente, canal, volumen o promociones, manteniendo una trazabilidad completa de cada cambio. En esencia, este contexto proporciona flexibilidad para manejar distintos **listas de precios** y asegurar que siempre se aplique el importe adecuado en cada venta.

## 1.9 Publicación de Eventos (Outbound) ↗

Los eventos **salientes** que **ListaPreciosBC** publica son consumidos por otros contextos para enterarse de las decisiones de precios y cambios relevantes en este dominio:

Evento Publicado (Destino)	Propósito / Uso
<b>PrecioAplicado</b>   ( <i>ComprobantesElectonicosBC</i> , <i>IndicadoresNegocioBC</i> )	Se genera al aplicar un precio específico durante la emisión de un comprobante. Informa qué lista y precio fueron utilizados en la transacción. <i>ComprobantesElectonicosBC</i> puede usarlo para fines de <b>auditoría</b> de facturación (registro del precio aplicado), mientras que <i>IndicadoresNegocioBC</i> lo consume para actualizar sus <b>dashboards</b>

	<b>de márgenes y variaciones de precios en tiempo real.</b>
<b>ListaPrecioClienteModificada &lt;br/&gt;</b> <i>(GestionClientesBC)</i>	Evento emitido cuando se crea o cambia una lista de precios especial asociada a un cliente específico. Permite a <i>GestionClientesBC</i> sincronizar la información de segmentación o atributos del cliente con respecto a sus precios especiales. Por ejemplo, el área de gestión de clientes puede marcar a un cliente como “con precios personalizados” o ajustar su categoría en función de la existencia de una lista especial.

## 1.10 Restricciones y Reglas del Negocio

Las siguientes restricciones y reglas aseguran la coherencia de las listas de precios:

- **Restricción de unicidad:** No puede haber dos **listas de precios vigentes** con el mismo criterio y para el mismo producto. Esto evita conflictos de aplicación.
- **Consistencia de tramos:** Los rangos en **RangoVolumen** deben ser válidos y no superponerse (**cantidadMin < cantidadMax**). Asimismo, el **PeriodoVigencia** debe cumplir **fechaInicio < fechaFin**.
- **Prioridad de evaluación (RN-LP-001):** Las listas se evalúan en orden fijo: **CLIENTE > CANAL > VOLUMEN > PROMOCIÓN > BASE**. Esto determina qué lista prevalece cuando múltiples listas aplican al mismo producto.
- **Periodo coherente (RN-LP-002):** Cada **PeriodoVigencia** en las listas debe ser lógico (**fechaInicio anterior a fechaFin**).
- **Valor positivo (RN-LP-003):** Todo **PrecioEspecifico.valor** debe ser mayor que cero (no se permiten precios negativos o nulos).
- **Moneda uniforme (RN-LP-004):** Todos los precios dentro de una misma lista deben compartir la misma moneda.
- **Soft-delete seguro (RN-LP-005):** Sólo se puede inhabilitar (eliminar lógicamente) una lista si no existen transacciones activas asociadas a ella (p.ej. facturas no anuladas). Esto garantiza integridad histórica.

Estas reglas se aplican automáticamente en los *use cases*. Por ejemplo, al crear o editar una lista, el sistema valida que no se dupliquen listas vigentes (aplicando RN-LP-001) y que los rangos de volumen y fechas sean consistentes.

# Aggregates, Entities, VO, Domains Events, Policies & Specifications\_lp

## 1. Entidades

**Descripción general:** Una **Entidad** es un objeto con identidad propia (ID) que evoluciona a lo largo del tiempo y tiene ciclo de vida relevante en el dominio.

Entidad	Descripción	Comportamiento	Relaciones
<b>ListaPrecio</b>	Agregado raíz que agrupa criterios y precios específicos.	<code>agregarPrecioEspecifico()</code> , <code>actualizarPrecio()</code> , <code>inhabilitarLista()</code>	1:∞n PrecioEspecifico, 1:N HistorialPrecio
<b>PrecioEspecifico</b>	Precio para un criterio dado: valor, moneda, prioridad, vigencia.	<code>validarValor()</code> , <code>aplicarPrioridad()</code> , <code>esVigente()</code>	N:1 ListaPrecio
<b>HistorialPrecio</b>	Registro inmutable de cada cambio de un <b>PrecioEspecifico</b> .	<code>registrarCambio()</code> , <code>obtenerDetallesCambio()</code>	N:1 PrecioEspecifico
<b>CriterioLista</b>	Representa formato de criterio: cliente, canal, volumen o promoción.	<code>cumpleCriterio(clienteId, canal, cantidad, fecha)</code>	1:N PrecioEspecifico

---

## 2. Objetos de Valor (Value Objects)

**Descripción general:** Los **Value Objects** son inmutables y se comparan por su valor, encapsulan atributos que describen propiedades sin identidad propia.

VO	Descripción	Atributos / Valores posibles	Uso / Importancia
<b>TipoLista</b>	Enum que clasifica la lista: CLIENTE, CANAL, VOLUMEN, PROMOCION, BASE.	Enumerado fijo.	Define orden de prioridad y lógica de aplicación.
<b>Moneda</b>	VO que indica la moneda ( PEN , USD ).	Enumerado de monedas.	Determina cálculos y consistencia de moneda.

RangoVolumen	Rango de cantidades {min, max}.	cantidadMin , cantidadMax .	Para descuentos por volumen.
PeriodoVigencia	Rango de fechas {inicio, fin}.	fechaInicio , fechaFin .	Para promociones temporales.
Prioridad	Nivel numérico de evaluación (menor = mayor prioridad).	Valor entero positivo.	Controla orden de aplicación de listas.

### 3. Eventos de Dominio

**Definición y características:** Un **Domain Event** es un evento inmutable que señala la ocurrencia de un hecho relevante en el dominio.

#### Eventos clave:

1. ListaPrecioCreada
2. PrecioEspecificoAgregado
3. PrecioEspecificoModificado
4. ListaPrecioInhabilitada
5. PrecioAplicado

#### Detalle de eventos

Evento	Origen	Datos contenidos	Consumidores Potenciales
ListaPrecioCreada	Use Case Configurar Lista	listaPrecioId , tipoLista , usuarioId , fechaCreacion	UI, IndicadoresNegocioBC, ComprobantesElectronico sBC
PrecioEspecificoAgregado	Agregar precio a Lista	precioEspecificoId , listaPrecioId , valor , criterio	UI, IndicadoresNegocioBC
PrecioEspecificoModificado	Use Case Editar Precio	precioEspecificoId , camposModificados , fechaModificacion	HistorialProductoBC, UI
ListaPrecioInhabilitada	Use Case Inhabilitar Lista	listaPrecioId , usuarioId , motivo , fechaInhabilitacion	ComprobantesElectronico sBC, UI

PrecioAplicado	Emisión de Comprobante	listaPrecioId , precioEspecificoId , comprobanteId , monto	IndicadoresNegocioBC, Auditoría
----------------	------------------------	---	------------------------------------

#### 4. Políticas (Policies)

**Descripción general:** Las **Policies** encapsulan reglas de negocio complejas y centralizadas, aplicadas durante casos de uso.

Código	Regla / Descripción breve
RN-LP-001	Validar unicidad de <code>ListaPrecio</code> por criterio y producto.
RN-LP-002	<code>PeriodoVigencia.inicio &lt; PeriodoVigencia.fin</code> .
RN-LP-003	<code>RangoVolumen.min &lt; RangoVolumen.max</code> y rangos no solapados.
RN-LP-004	<code>Prioridad</code> dentro de rango permitido (1-10).
RN-LP-005	Un solo registro vigente por criterio en cada momento.

#### Cuadro de control de reglas

Rango de Códigos	Entidad / Tema	Descripción del Grupo	Total Definidas	Próx. Código	Observaciones
RN-LP-001-RN-LP-005	ListaPreciosBC	Unicidad, vigencia, rangos y prioridades.	5	RN-LP-006	Incluir reglas de permisos y eventos.

#### 5. Especificaciones (Specifications)

**Descripción general:** Las **Specifications** son predicados reutilizables para evaluar si una entidad o VO cumple criterios de negocio.

Caso	Precondiciones	Reglas Aplicadas
EsListaVigente	<code>listaPrecioId</code> existe	RN-LP-005
EsPeriodoValido	<code>PeriodoVigencia</code> definido	RN-LP-002

EsVolumenValido	RangoVolumen.min < RangoVolumen.max	RN-LP-003
EsPrioridadValida	Prioridad entre 1 y 10	RN-LP-004
EsListaUnicaPorCriterioProducido	No existe otra lista vigente igual	RN-LP-001

# Casos de Uso\_lp

## 1. ¿Qué son los Casos de Uso en DDD?

En DDD, los **Casos de Uso** documentan las interacciones entre actores y el sistema para lograr objetivos de negocio. Cada caso de uso define:

- **Actor Principal**
- **Descripción** de la acción en términos de negocio
- **Precondiciones** necesarias
- **Flujo Principal** (paso a paso)
- **Flujos Alternativos** (desvíos)
- **Eventos Generados** (Domain Events)
- **Entidades Afectadas** (si aplica)

## 2. Resumen de Casos de Uso

Nº	Caso de Uso	Actor Principal
1	Configurar Lista de Precios	Usuario del sistema
2	Agregar Precio Específico	Usuario del sistema
3	Modificar Precio Específico	Usuario del sistema
4	Inhabilitar Lista de Precios	Usuario del sistema
5	Obtener Precio para Comprobante	Subsistema Facturación
6	Importar Listas desde CSV	Usuario del sistema
7	Exportar Listas a CSV	Usuario del sistema
8	Notificar Cambios de Lista	Sistema de Notificaciones

## 3. Detalle de cada Caso de Uso

### Caso de Uso 1 – Configurar Lista de Precios

- **Actor Principal:** Usuario del sistema
- **Descripción:** Crear una nueva `ListaPrecio` con criterios y vigencia.

#### Precondiciones:

- Usuario autenticado y con permiso `ListaPrecios.Crear`.
- `criterio` (cliente, canal, volumen o promoción) válido.

#### **Flujo Principal:**

1. El actor proporciona: `tipoLista`, `criterioLista`, `moneda`, `prioridad`, `periodoVigencia`.
2. El sistema valida que no exista otra lista vigente con mismo criterio ( RN-LP-001 ).
3. Crea entidad `ListaPrecio` y registra `fechaCreacion`.
4. Publica evento `ListaPrecioCreada`.
5. Retorna `listaPrecioId`.

#### **Flujos Alternativos:**

- Criterio duplicado → lanza `ListaPrecioDuplicadaException`.
- Periodo inválido → lanza `RangoVigenciaInvalidoException`.

#### **Eventos Generados:**

- `ListaPrecioCreada`

#### **Entidades Afectadas:**

- `ListaPrecio`



---

#### Caso de Uso 2 – Agregar Precio Especifico

- **Actor Principal:** Usuario del sistema
- **Descripción:** Añadir un `PrecioEspecifico` a una lista existente.

#### **Precondiciones:**

- `listaPrecioId` existe y está `VIGENTE`.
- `criterioPrecio` (rangoVolumen o periodoVigencia) válido.

#### **Flujo Principal:**

1. Actor envía: `listaPrecioId`, `valor`, `criterioPrecio`.
2. Sistema valida vigencia de la lista y formato de VO ( RN-LP-003 , RN-LP-002 ).
3. Agrega `PrecioEspecifico` y registra en `HistorialPrecio`.
4. Publica evento `PrecioEspecificoAgregado`.
5. Confirma operación.

#### **Flujos Alternativos:**

- Lista inactiva → `InvalidStateException`.
- Valor  $\leq 0$  → `PrecioInvalidoException`.

#### **Eventos Generados:**

- `PrecioEspecificoAgregado`

#### **Entidades Afectadas:**

- `PrecioEspecifico`, `HistorialPrecio`



### Caso de Uso 3 – Modificar Precio Especifico

- **Actor Principal:** Usuario del sistema
- **Descripción:** Actualizar atributos de un `PrecioEspecifico` existente.

#### Precondiciones:

- `precioEspecificoId` existe y pertenece a lista `VIGENTE`.

#### Flujo Principal:

1. Actor suministra: `precioEspecificoId`, `camposModificados`.
2. Sistema valida permiso `ListaPrecios.Editar` y formatos de VO.
3. Aplica cambios y registra en `HistorialPrecio`.
4. Publica evento `PrecioEspecificoModificado`.
5. Devuelve confirmación.

#### Flujos Alternativos:

- Precio no existe → `EntityNotFoundException`.
- Formato inválido → excepciones de validación.

#### Eventos Generados:

- `PrecioEspecificoModificado`

#### Entidades Afectadas:

- `PrecioEspecifico`, `HistorialPrecio`



---

### Caso de Uso 4 – Inhabilitar Lista de Precios

- **Actor Principal:** Usuario del sistema
- **Descripción:** Marcar una `ListaPrecio` como `EXPIRADA`.

#### Precondiciones:

- `listaPrecioId` existe.
- No existen transacciones activas usando la lista ( `RN-LP-005` ).

#### Flujo Principal:

1. Actor envía: `listaPrecioId`, `motivoInhabilitacion`.
2. Sistema valida ausencia de uso en comprobantes activos.
3. Cambia `estado` a `EXPIRADA`, registra en `HistorialPrecio`.
4. Publica evento `ListaPrecioInhabilitada`.
5. Confirma operación.

#### Flujos Alternativos:

- Transacciones activas → `DependenciaActivaException`.

#### Eventos Generados:

- `ListaPrecioInhabilitada`

#### Entidades Afectadas:

- `ListaPrecio`



---

#### Caso de Uso 5 – Obtener Precio para Comprobante

- **Actor Principal:** Subsistema Facturación
- **Descripción:** Seleccionar el `PrecioEspecifico` más adecuado para un comprobante.

#### Precondiciones:

- `clienteId`, `productoServicioId`, `cantidad` y `fecha` provistos.

#### Flujo Principal:

1. Sistema consulta `CatalogoArticulosBC` para validar producto.
2. Invoca `ListaPreciosBC.ObtenerPrecio(...)`.
3. Recorre listas en orden RN-LP-001: CLIENTE > CANAL > VOLUMEN > PROMOCION > BASE.
4. Retorna `precioFinal` y publica `PrecioAplicado`.

#### Flujos Alternativos:

- No existe lista → usa precio base.
- Error criterio → lanza `PrecioNoEncontradoException`.

#### Eventos Generados:

- `PrecioAplicado`

#### Entidades Afectadas:

- Ninguna (solo lectura)



---

#### Caso de Uso 6 – Importar Listas desde CSV

- **Actor Principal:** Usuario del sistema
- **Descripción:** Carga masiva de `ListaPrecio` y `PrecioEspecifico` desde archivo.

#### Precondiciones:

- Archivo CSV con columnas obligatorias (`tipoLista`, `criterio`, `valor`, `moneda`, `vigencia`).

#### Flujo Principal:

1. Actor sube archivo.
2. Sistema valida encabezados y estructura.
3. Procesa cada fila:
  - a. Si lista no existe → Caso de Uso 1.
  - b. Si existe → Caso de Uso 2 o 3 según `valor` nuevo.
4. Genera resumen de éxitos y errores.
5. Publica eventos `ListaPrecioCreada` / `PrecioEspecificoAgregado` por fila.

#### **Flujos Alternativos:**

- Fila inválida → omitir y registrar en resumen.

#### **Eventos Generados:**

- `ListaPrecioCreada` , `PrecioEspecificoAgregado`

#### **Entidades Afectadas:**

- `ListaPrecio` , `PrecioEspecifico`



---

#### Caso de Uso 7 – Exportar Listas a CSV

- **Actor Principal:** Usuario del sistema
- **Descripción:** Generar archivo con todas las listas y precios para edición externa.

#### **Precondiciones:**

- Filtros opcionales válidos.

#### **Flujo Principal:**

1. Actor selecciona filtros (tipoLista, estado, fecha).
2. Sistema recupera datos y construye CSV.
3. Ofrece descarga.

#### **Flujos Alternativos:**

- Sin datos → retorna CSV vacío con encabezados.

#### **Eventos Generados:**

- Ninguno

#### **Entidades Afectadas:**

- Ninguna



---

#### Caso de Uso 8 – Notificar Cambios de Lista

- **Actor Principal:** Sistema de Notificaciones
- **Descripción:** Enviar alertas al equipo comercial tras cambios críticos.

#### **Precondiciones:**

- Suscripción activa a eventos de `ListaPrecioCreada` o `Inhabilitada` .

#### **Flujo Principal:**

1. Al recibir evento, notificador construye mensaje (listaNombre, tipoEvento, usuario, fecha).
2. Envía correo/SMS al grupo de usuarios con permiso `VER_PRECIOS` .

**Flujos Alternativos:**

- Falla envío → registra en cola de reintentos.

**Eventos Generados:**

- Ninguno (es reacción)

**Entidades Afectadas:**

- Ninguna



**4. Observaciones** 

- Cada caso de uso debe implementarse como **Application Service** separado.
- Las excepciones específicas ( `ListaPrecioDuplicadaException` , etc.) facilitan pruebas e integraciones.
- Documentar flujos alternativos en tests de integración para cobertura completa.

# Eventos de Dominio\_lp

## 1. ¿Qué es un Evento de Dominio?

Un **Evento de Dominio** es una notificación inmutable que refleja un hecho relevante ya ocurrido en el dominio. Sirve para:

- **Capturar** cambios de estado o acciones clave (e.g., creación, modificación, inhabilitación).
- **Desacoplar** componentes: publican sin invocar directamente a consumidores.
- **Auditar y reaccionar** de forma asincrónica.

### Características clave:

- Se **genera** al completar un caso de uso.
- Es **inmutable**: sus datos permanecen constantes.
- Facilita **consistencia eventual** y reacciones **desacopladas**.

---

## 2. Resumen de eventos de dominio

#	Evento	Descripción Corta
1	ListaPrecioCreada	Nueva lista de precios creada.
2	PrecioEspecificoAgregado	Se agregó un precio específico a una lista.
3	PrecioEspecificoModificado	Un precio específico fue actualizado.
4	ListaPrecioInhabilitada	Una lista de precios fue marcada como expirada.
5	PrecioAplicado	Se aplicó un precio al emitir un comprobante.

---

## 3. Detalle de cada Evento de Dominio

### Evento de Dominio: ListaPrecioCreada

Campo	Tipo	Descripción
listaPrecioId	UUID	Identificador único de la lista creada.

<code>tipoLista</code>	Enum	CLIENTE, CANAL, VOLUMEN, PROMOCION o BASE.
<code>criterioLista</code>	VO	Parámetro de segmentación (clienteId, canal, etc.).
<code>usuarioId</code>	UUID	ID del usuario que creó la lista.
<code>fechaCreacion</code>	Timestamp	Momento en que se generó el evento.

**Origen:** Use Case **Configurar Lista de Precios**.

**Consumidores:** ComprobantesElectonicosBC (validación), IndicadoresNegocioBC (métricas), UI Admin (visualizar).

**Observaciones:** Indexar para búsquedas rápidas.

Evento de Dominio: `PrecioEspecificoAgregado` 

Campo	Tipo	Descripción
<code>precioEspecificoId</code>	UUID	Identificador del precio añadido.
<code>listaPrecioId</code>	UUID	Lista a la que se agregó el precio.
<code>valor</code>	Decimal	Monto del precio.
<code>criterioPrecio</code>	VO	RangoVolumen o PeriodoVigencia aplicable.
<code>usuarioId</code>	UUID	ID del usuario responsable.
<code>fechaCreacion</code>	Timestamp	Fecha de creación del precio.

**Origen:** Caso de Uso **Agregar Precio Específico**.

**Consumidores:** IndicadoresNegocioBC (reportes), HistorialPrecio (registro), UI Admin.

**Observaciones:** Verificar reglas de prioridad tras agregar.

Evento de Dominio: `PrecioEspecificoModificado` 

Campo	Tipo	Descripción
<code>precioEspecificoId</code>	UUID	Precio modificado.

<code>camposModificados</code>	List<{campo, anterior, nuevo}>	Cambios realizados.
<code>usuarioId</code>	UUID	ID del usuario que modificó.
<code>fechaModificacion</code>	Timestamp	Momento del cambio.

**Origen:** Caso de Uso **Modificar Precio Especifico**.

**Consumidores:** HistorialPrecio (almacenar versión), UI Admin (refrescar datos).

**Observaciones:** Debe disparar recálculo de caché y notificaciones si prioridad cambia.

Evento de Dominio: `ListaPrecioInhabilitada` 

Campo	Tipo	Descripción
<code>listaPrecioId</code>	UUID	Lista que se inhabilitó.
<code>usuarioId</code>	UUID	ID del usuario responsable.
<code>motivoInabilitacion</code>	String	Descripción del motivo.
<code>fechaInabilitacion</code>	Timestamp	Fecha de expiración.

**Origen:** Use Case **Inhabilitar Lista de Precios**.

**Consumidores:** ComprobantesElectonicosBC (rechazo de lista), UI Admin.

**Observaciones:** Asegurar que no queden listas vivas con misma prioridad.

Evento de Dominio: `PrecioAplicado` 

Campo	Tipo	Descripción
<code>listaPrecioId</code>	UUID	Lista usada para cálculo.
<code>precioEspecificoId</code>	UUID	Precio aplicado.
<code>comprobanteId</code>	UUID	ID del comprobante.
<code>monto</code>	Decimal	Importe resultante.
<code>fechaAplicacion</code>	Timestamp	Fecha y hora de aplicación.

**Origen:** Caso de Uso **Obtener Precio para Comprobante**.

**Consumidores:** IndicadoresNegocioBC (variación de precios), Auditoría.

**Observaciones:** Utilizado para reconstruir historial de precios.

#### **4. Observaciones Generales**

- Publicar eventos en Event Bus o sistema de mensajería (Kafka, RabbitMQ).
- Versionar contratos de eventos para mantener compatibilidad.
- Documentar eventos en el repositorio de especificaciones para integradores.

# Restricciones y Reglas de Negocio\_lp

## 1. Contexto

En ListaPreciosBC, las **Reglas de Negocio** y **Restricciones** aseguran que las listas de precio cumplan criterios de unicidad, coherencia temporal y requisitos operativos antes de crear, modificar o aplicar tarifas.

## 2. Definiciones

- **Reglas de Negocio:** Condiciones lógicas o normativas que definen cómo debe comportarse el sistema durante la ejecución de los casos de uso. Su incumplimiento provoca un rechazo o registro de auditoría.
- **Restricciones:** Limitaciones operativas o técnicas que impiden iniciar una acción si no se cumplen criterios previos. Se aplican antes o al inicio del caso de uso y bloquean la ejecución.

## 3. Diferencia entre Reglas y Restricciones

Concepto	Rol en el sistema	Se aplica cuando...	Consecuencia si no se cumple
Regla de Negocio	Define comportamiento en flujos de uso	Durante la ejecución de un caso de uso	Rechaza la acción o audita el evento
Restricción	Define cuándo NO debe permitirse una acción	Antes o al inicio del caso de uso	Bloquea la ejecución con error específico

---

## 4. Condiciones de negocio por Caso de Uso

### Caso de Uso 1 – Configurar Lista de Precios

- **Entidad base:** ListaPrecio
- **Rango de códigos:** RN-LP-001 a RN-LP-002

#### Restricciones:

- No puede existir otra lista **VIGENTE** con el mismo `criterioLista` y producto.
- Usuario con permiso `ListaPrecios.Crear`.

#### Reglas de Negocio Aplicadas:

Código	Regla
RN-LP-001	Validar unicidad de lista por criterio y producto.

RN-LP-002	PeriodoVigencia.inicio < PeriodoVigencia.fin .
-----------	--

#### Caso de Uso 2 – Agregar Precio Especifico

- **Entidad base:** PrecioEspecifico
- **Rango de códigos:** RN-LP-003

##### Restricciones:

- La `ListaPrecio` debe estar en estado **VIGENTE**.

##### Reglas de Negocio Aplicadas:

Código	Regla
RN-LP-003	<code>PrecioEspecifico.valor &gt; 0</code> y <code>Moneda</code> válida.

#### Caso de Uso 3 – Modificar Precio Especifico

- **Entidad base:** PrecioEspecifico
- **Rango de códigos:** RN-LP-003, RN-LP-004

##### Restricciones:

- El `PrecioEspecifico` debe existir y pertenecer a una lista **VIGENTE**.

##### Reglas de Negocio Aplicadas:

Código	Regla
RN-LP-003	<code>PrecioEspecifico.valor &gt; 0.</code>
RN-LP-004	<code>Prioridad</code> dentro del rango permitido (1–10).

#### Caso de Uso 4 – Inhabilitar Lista de Precios

- **Entidad base:** ListaPrecio
- **Rango de códigos:** RN-LP-005

##### Restricciones:

- No debe haber comprobantes activos referenciando la lista.

##### Reglas de Negocio Aplicadas:

Código	Regla

RN-LP-005	Solo permitir soft-delete si no hay transacciones activas.
-----------	--

#### Caso de Uso 5 – Obtener Precio para Comprobante

- **Entidad base:** Operación de cálculo de precio
- **Rango de códigos:** RN-LP-001, RN-LP-006

#### Restricciones:

- Debe existir al menos una lista (incluyendo tipo **BASE**) para el producto.

#### Reglas de Negocio Aplicadas:

Código	Regla
RN-LP-001	Prioridad de evaluación: CLIENTE > CANAL > VOLUMEN > PROMOCION > BASE.
RN-LP-006	Si no se encuentra lista, usar precio base.

#### 5. Cuadro de Control de Códigos de Reglas de Negocio

Rango de Códigos	Entidad / Tema	Descripción del Grupo	Total Definidas	Próx. Código	Observaciones
RN-LP-001–RN-LP-006	ListaPrecios BC	Unicidad, vigencia, valor y prioridad	6	RN-LP-007	Reservado para reglas de importación masiva

## Lenguaje Ubicuo\_lp

En *ListaPreciosBC* el lenguaje compartido define comandos de aplicación, eventos de dominio, entidades y objetos de valor claves. Por ejemplo, los **comandos** pueden llamarse `CrearListaPrecioCommand`, `AgregarPrecioEspecificoCommand`, `ModificarPrecioCommand` o `InhabilitarListaPrecioCommand`. Cada comando activa un caso de uso orquestado por un servicio de aplicación. Al ejecutarlos, se publican **eventos de dominio** como *ListaPrecioCreada*, *PrecioEspecificoAgregado*, *PrecioEspecificoModificado*, *ListaPrecioInhabilitada* o *PrecioAplicado*. Estos eventos inmutables señalan cambios relevantes para otros BC (e.g. *PrecioAplicado* informa el precio usado en cada factura).

Las **entidades** principales son el agregado raíz *ListaPrecio* (colección de entradas tarifarias bajo un criterio), la entidad *PrecioEspecifico* (precio concreto para un criterio dado: valor, moneda, prioridad, vigencia), y *HistorialPrecio* (registro inmutable de cambios en un *PrecioEspecifico*). Como **Objetos de Valor** (VO) se usan los criterios de aplicación: *TipoLista* (enum: CLIENTE, CANAL, VOLUMEN, PROMOCION, BASE), *Moneda* (enum: PEN, USD), *RangoVolumen* (rangos {min, max} para descuentos por cantidad), *PeriodoVigencia* (fechas {inicio, fin} de promoción), y el campo *Prioridad* numérico (orden de evaluación). Además existe *CriterioLista* (que puede incluir cliente, canal, volumen o promoción).

Por ejemplo, un JSON de entrada para crear una lista de precios podría ser:

```
1 POST /listas-precios
2 Content-Type: application/json
3
4 {
5     "clienteId": "C54321",
6     "moneda": "PEN",
7     "periodoVigencia": {
8         "fechaFin": "2025-07-31",
9         "fechaInicio": "2025-07-01"
10    },
11    "prioridad": 1,
12    "tipoLista": "CLIENTE"
13 }
```

La respuesta devolvería el `listaPrecioId` generado. A su vez, al emitir un comprobante se invoca algo como:

```
1 GET /listas-precios/obtenerPrecio?clienteId=C54321&productoId=P1001&cantidad=10&fecha=2025-07-01
2 Host: api.facturafacil.com
```

que responde

```
1 {
2     "listaPrecioId": "L-uuid",
3     "precio": 250.00,
4     "precioEspecificoId": "P-uuid"
5 }
```

Estos usos ilustran cómo el BC selecciona automáticamente el precio correcto para cada factura.

Elemento	Descripción
<b>ListaPrecio</b>	Agregado raíz que agrupa criterios y precios específicos.
<b>PrecioEspecifico</b>	Entidad con precio (valor, moneda, prioridad, vigencia) para un criterio dado.
<b>HistorialPrecio</b>	Registro inmutable de cada modificación de un <i>PrecioEspecifico</i> .
<b>CriterioLista (VO)</b>	Formato de criterio de aplicación: cliente, canal, volumen o promoción.
<b>TipoLista (VO)</b>	Enum (CLIENTE, CANAL, VOLUMEN, PROMOCION, BASE) que define prioridad de aplicación.
<b>Moneda (VO)</b>	Enum (PEN, USD) que fija la moneda de los precios.
<b>RangoVolumen (VO)</b>	Rango de cantidades <code>{min, max}</code> para descuentos por volumen.
<b>PeriodoVigencia (VO)</b>	Rango de fechas <code>{inicio, fin}</code> para listas/promociones temporales.

## Servicios de Aplicación\_lp

Los servicios de aplicación orquestan los casos de uso de negocio. Cada servicio expone interfaces (métodos o endpoints REST) que reciben parámetros claros y devuelven respuestas (IDs, status, datos). Ejemplos de métodos podrían ser:

- `crearListaPrecio(tipoLista, criterio, moneda, prioridad, periodoVigencia) → listaPrecioId` – Valida reglas de negocio (e.g. RN-LP-001) y crea un *ListaPrecio*. Publica el evento *ListaPrecioCreada* y retorna el ID.
- `agregarPrecioEspecifico(listaPrecioId, valor, criterioPrecio) → precioEspecificoId` – Verifica vigencia y unicidad; agrega un nuevo *PrecioEspecifico* al agregado y guarda en historial. Publica *PrecioEspecificoAgregado* cuando se guarda.
- `modificarPrecioEspecifico(precioEspecificoId, camposModificados) → void` – Actualiza atributos permitidos de un *PrecioEspecifico*, registra en historial y emite *PrecioEspecificoModificado*.
- `inhabilitarListaPrecio(listaPrecioId, usuarioId, motivo) → void` – Marca la lista como inactiva (soft-delete) tras comprobar que cumple RN-LP-005 (no transacciones activas). Publica *ListaPrecioInhabilitada*.
- `obtenerPrecio(clienteId, productoId, cantidad, fecha) → {precio, listaPrecioId, precioEspecificoId}` – Busca la mejor lista aplicable según prioridad (CLIENTE > CANAL > VOLUMEN > PROMOCION > BASE) y retorna el precio resultante al subsistema de facturación.
- `importarListas(preciosCSV) / exportarListas()` – Procesan carga/descarga masiva desde archivos Excel.
- `notificarCambioLista(listaPrecioId)` – Envía alertas al equipo comercial tras cambios importantes.

Cada método se implementa en un *Aplicacion Service* que inyecta repositorios y servicios de dominio. Por ejemplo, el flujo principal de crear lista es:

1. El usuario (actor) envía (`tipoLista, criterio, moneda, prioridad, periodoVigencia`).
2. El servicio valida reglas (RN-LP-001 a RN-LP-005).
3. Crea el agregado *ListaPrecio*, guarda fecha de creación en *HistorialPrecio*.
4. Publica evento *ListaPrecioCreada* y retorna `listaPrecioId`.

**Ejemplo de JSON de servicio:** Crear lista de precios.

```
1 POST /api/listas-precios
2 Content-Type: application/json
3
4 {
5   "clienteId": "C54321",
6   "moneda": "PEN",
7   "periodoVigencia": {
8     "fechaFin": "2025-07-31",
9     "fechaInicio": "2025-07-01"
10 },
11   "prioridad": 1,
12   "tipoLista": "CLIENTE"
13 }
```

## Response

```
1  {
2    "listaPrecioId": "fe0211b3-2a4f-11ec-8f27-0242ac120002"
3 }
```

## Infraestructura y Adaptadores\_lp

La infraestructura implementa los patrones de persistencia, integración y mensajería. Se define un **Repositorio** (e.g. `RepositorioListaPrecios`) que abstrae la persistencia de los agregados. Por ejemplo, `RepositorioListaPrecios` usa una base de datos relacional o NoSQL para guardar `ListaPrecio` y sus entidades hijas. Se usan *mappers* o *factories* para reconstruir entidades desde datos primitivos, y *handlers* para responder a eventos de dominio (por ejemplo, actualizar cálculo tras un evento de `PrecioEspecificoModificado`).

Para la integración con otros BC se emplea un bus de eventos (EventBus) o cola de mensajes. Los eventos entrantes (*Inbound*, p. ej. `ComprobanteEmitido`, `ProductoModificado`, `ClienteModificado`, `ParametrosPreciosActualizados`) son manejados por adaptadores que traducen (mapean) los datos externos al modelo interno de `ListaPreciosBC`. Por ejemplo, al recibir un evento `ClienteModificado` de **GestionClientesBC**, un **Event Handler** invoca un servicio de dominio para actualizar la segmentación de ese cliente en las listas internas. De modo similar, un evento `ProductoModificado/Inhabilitado` de **CatalogoArticulosBC** actualiza las listas que usan ese SKU.

La comunicación saliente (*Outbound*) ocurre mediante publicación de eventos de dominio: `PrecioAplicado` se envía al bus para que **IndicadoresNegocioBC** actualice métricas, y `ListaPrecioInhabilitada` puede notificar a **GestionClientesBC** que ajuste segmentaciones especiales. Si se integra con dominios externos (p. ej. Identidad y Autenticación), se usa un **Anti-Corruption Layer (ACL)**: un adaptador traduce el modelo externo al VO interno `UsuarioRegistrado`, aislando así el modelo de autenticación de Factura Fácil.

Además se aplican patrones de construcción comunes: un *Factory* para crear nuevas instancias de `ListaPrecio` con invariante satisfecha, un *Command Handler* para cada comando de aplicación, y *Specification* para validar las reglas RN-LP (e.g. `EsListaUnicaPorCriteriaProducto`). En resumen, la capa de infraestructura provee repositorios, adaptadores de mensajería (EventBus), gateways API (para identidades) y los patrones Factory/Handler/Mapper necesarios para encajar el dominio en la arquitectura técnica.

## Mapa de Contextos\_lp

El **contexto de ListaPrecios** interactúa explícitamente con otros BC oficiales de Factura Fácil:

**ComprobantesElectronicosBC, CatalogoArticulosBC, GestionClientesBC, ConfiguracionSistemaBC,**

**IndicadoresNegocioBC, e Identidad y Autenticación.** A continuación se ilustran las relaciones y patrones DDD:

- **ComprobantesElectronicosBC → ListaPreciosBC:** Al emitir un comprobante, este subsistema llama a *ListaPreciosBC* (p. ej. vía llamada RPC o API REST) para calcular el precio correcto. Aquí *Comprobantes* es *cliente* y *ListaPrecios* es *proveedor* (Customer–Supplier). Se podría implementar como un **Open Host Service**: *ListaPreciosBC* expone un servicio `obtenerPrecio(...)` que *Comprobantes* consume. Después, *Comprobantes* publica eventos como *ComprobanteEmitido* (que *ListaPreciosBC* también puede consumir para auditoría).
- **CatalogoArticulosBC → ListaPreciosBC:** El catálogo de productos publica eventos (ej. *ProductoModificado*, *ProductoInhabilitado*) que *ListaPreciosBC* consume. Se utiliza un **Published Language**: *ListaPrecios* adopta los eventos tal cual sin transformar su modelo interno. No hay lógica compartida más allá del contrato de eventos compartido.
- **GestionClientesBC → ListaPreciosBC:** De forma similar, *ClienteModificado* se publica cuando cambia el segmento o estado de un cliente, y *ListaPreciosBC* lo consume para ajustar listas por cliente. Este vínculo puede verse como **Customer–Supplier**, donde *GestionClientesBC* suministra datos de cliente y *ListaPreciosBC* los utiliza.
- **ConfiguracionSistemaBC → ListaPreciosBC:** Cuando se actualizan parámetros globales (moneda por defecto, reglas de redondeo, límites de tramos), *ConfiguracionSistemaBC* emite un evento *ParametrosPreciosActualizados* que *ListaPreciosBC* consume. Aquí basta con un **Published Language**, ya que se envían datos simples de configuración.
- **ListaPreciosBC → IndicadoresNegocioBC:** *ListaPreciosBC* publica el evento *PrecioAplicado* cada vez que se calcula un precio para un comprobante. *IndicadoresNegocioBC* lo consume para actualizar dashboards de márgenes y variaciones. Este es un caso clásico de patrón **Customer–Supplier**: *ListaPrecios* es proveedor de datos, *Indicadores* consumidor.
- **ListaPreciosBC → GestionClientesBC:** Al cambiar listas por segmento, *ListaPreciosBC* publica (p.ej.) *ListaPrecioClienteModificada*, que *GestionClientesBC* usa para sincronizar información de clientes con precios especiales.
- **Identidad y Autenticación (externo) → ListaPreciosBC:** Factura Fácil integra los usuarios con un servicio centralizado de identidades. Se aplica **Anti-Corruption Layer (ACL)**: la información de registro/login se traduce vía adaptador a un VO interno (*UsuarioRegistrado*), evitando contaminar el modelo de precios con la estructura de acceso externa.
- **Patrones adicionales:** La integración con el *API Service de Facturación* (SUNAT) descrita en la documentación usa el patrón **Conformist**, pero esto ocurre principalmente en **ComprobantesElectrónicosBC**. En el contexto de *ListaPreciosBC* se aplican los patrones mencionados arriba (Customer–Supplier, Published Language, ACL) para asegurar coherencia con los BC oficiales.

Estos enlaces se resumen en el siguiente diagrama de contexto:





## Política de Consistencia y Transaccionalidad\_lp

El diseño favorece la **consistencia eventual** y uso de eventos para coordinación entre BCs. La mayoría de las integraciones (p. ej. ajustes de precios ante eventos de catálogo o cliente) se gestionan de forma asíncrona vía mensajería, evitando transacciones distribuidas. Por ejemplo, al emitir un comprobante se calcula el precio en *ListaPreciosBC* y luego se publica *PrecioAplicado*, procesado luego por otros servicios sin bloquear la transacción inicial.

Sin embargo, ciertas invariantes requieren validaciones de consistencia. La regla RN-LP-005 establece que *no se puede soft-delete una lista si hay transacciones activas asociadas* (por ejemplo, facturas aún válidas con esa lista). Para garantizarlo se puede implementar un patrón de **Saga de orquestación**: al inhabilitar una lista, el servicio puede consultar síncronamente a **ComprobantesElectonicosBC** (o al contexto de ventas) si existen comprobantes pendientes con esa lista, abortando la operación en caso afirmativo. Alternativamente, se puede manejar eventualidad: emitir *ListaPrecioInhabilitada* y luego compensar (anular) si llega un evento de comprobante conflictivo. En ambos casos, la política general es preferir la **eventualidad** y diseñar flujos compensatorios, salvo en validaciones críticas donde se pueda hacer una comprobación inmediata.

En resumen, *ListaPreciosBC* funciona con consistencia eventual mediante eventos para comunicar cambios e integra validaciones estrictas mediante reglas de dominio. Las sagas (o transacciones distribuidas coordinadas) solo se usan cuando es imperativo asegurar reglas transversales (e.g. RN-LP-005), mientras que la mayoría de la orquestación se logra con eventos de dominio. De este modo se balancea la fiabilidad con la escalabilidad y flexibilidad del sistema, siguiendo los principios DDD establecidos en el documento oficial.

# Estrategia de Persistencia y Esquema de Datos\_lp

A continuación se describe cómo almacenar y mantener el modelo de **ListaPreciosBC** garantizando escalabilidad, aislamiento por tenant y compatibilidad con evoluciones de esquema.

## 11.1 Modelo físico de la entidad ↴

Se propone un esquema relacional compartido (multi-tenant en una sola base), con tres tablas principales:

Tabla	Columnas clave	Descripción
<b>lista_precio</b>	<code>id</code> (PK), <code>tenant_id</code> , <code>nombre</code> , <code>tipo_lista</code> , <code>criterio_json</code> , <code>activo</code> , <code>fecha_creacion</code> , <code>usuario_creo</code>	Agregado raíz. Cada registro define un conjunto de precios para un criterio (cliente, canal, etc.).
<b>precio_especifico</b>	<code>id</code> (PK), <code>lista_precio_id</code> (FK), <code>producto_id</code> , <code>valor</code> , <code>moneda</code> , <code>prioridad</code> , <code>cantidad_min</code> , <code>cantidad_max</code> , <code>fecha_inicio</code> , <code>fecha_fin</code> , <code>activo</code>	Entradas tarifarias dentro de cada lista. Incluye vigencia (volumen o periodo) y prioridad de aplicación.
<b>historial_precio</b>	<code>id</code> (PK), <code>precio_especifico_id</code> (FK), <code>valor_anterior</code> , <code>valor_nuevo</code> , <code>usuario_modifico</code> , <code>fecha_modificacion</code> , <code>motivo</code>	Registro inmutable de cada cambio en <code>precio_especifico</code> para auditoría.

### Notas sobre columnas JSON

- `criterio_json` en `lista_precio` puede almacenar en JSON el detalle del criterio (e.g. `{ "clienteId": "...", "canal": "ONLINE" }`).
- Esto permite añadir nuevos campos de criterio sin cambiar el esquema físico.

## 11.2 Índices recomendados y particionamiento por tenant ↗

### 1. Índices compuestos

- En `lista_precio` :

```
1 CREATE INDEX idx_lista_precio_tenant_activa
2   ON lista_precio (tenant_id, activo);
```

- En `precio_especifico` :

```
1 CREATE INDEX idx_precio_espec_tenant_vigencia
2   ON precio_especifico (lista_precio_id, activo, fecha_inicio, fecha_fin);
3 CREATE INDEX idx_precio_espec_producto
4   ON precio_especifico (producto_id, activo);
```

- En `historial_precio` :

```
1 CREATE INDEX idx_historial_precio_fk
2   ON historial_precio (precio_especifico_id);
3 CREATE INDEX idx_historial_precio_fecha
4   ON historial_precio (fecha_modificacion);
```

### Particionamiento por tenant

- Rango de partición sobre `lista_precio(tenant_id)` o sub-tablas por bloques de tenant (si hay cientos de tenants):

```
1 -- Ejemplo en PostgreSQL
2 CREATE TABLE lista_precio_p1 PARTITION OF lista_precio FOR VALUES IN
3 ('tenantA','tenantB',...);
```

- Alternativamente, usar **sharding** a nivel de base de datos si el volumen lo justifica.

### 2. Consideraciones de cache

- Mantener en memoria (Redis o similar) las listas y precios activos para cada tenant/producto para acelerar la resolución en tiempo real al emitir comprobantes.

---

## 11.3 Estrategia de versionado y migraciones de esquema ↗

### 1. Control de versiones de esquema

- Utilizar **Flyway** o **Liquibase** para gestionar cada cambio estructural.
- Versionar los scripts con prefijos secuenciales:

```
1 V1__crear_tablas_lista_precio.sql
2 V2__agregar_columna_criterio_json.sql
3 V3__crear_indices.sql
4 V4__particionar_tablas_por_tenant.sql
```

### 2. Migraciones en caliente

- Aplicar migraciones con **sin downtime**:
  - Crear nuevas columnas/tablas en paralelo.
  - Rellenar poco a poco datos históricos si se añade un VO.
  - Cambiar la aplicación para escribir ambos esquemas (viejo y nuevo).
  - Una vez estabilizado, eliminar el esquema antiguo.

### 3. Pruebas de migración

- Antes de producción, ejecutar migraciones en **entornos de staging** con datos representativos de todos los tenants.
- Incluir pruebas que verifiquen integridad referencial y correcta partición.

### 4. Rollback y compatibilidad

- En cada script, definir un **rollback** (cuando sea posible) para revertir cambios si se detectan problemas.
- Mantener compatibilidad con versiones anteriores de la aplicación durante un período de transición.

## Seguridad\_lp

La seguridad de **ListaPreciosBC** se basa en principios de arquitectura *microservicios secure by design*. Se recomienda:

- **Autenticación y autorización fuertes:** Usar OAuth2/OpenID Connect con tokens JWT para autenticar cada petición al API gateway. Luego, cada microservicio (incluyendo ListaPrecios) verifica el token y extrae roles o permisos. Por ejemplo, solo roles con permiso `ListaPrecios.Crear` pueden crear listas. Se implementan roles (p.ej. ADMIN, COMERCIAL) y permisos granulares (CREAR, EDITAR, VISUALIZAR, INHABILITAR) sobre recursos de lista de precios.
- **Canal seguro (TLS):** Toda comunicación externa e interna debe cifrarse vía HTTPS/TLS (incluso mTLS dentro del clúster). Esto evita intercepción de datos sensibles. Además, utilizar un *API Gateway* o proxy de ingreso que centralice la autenticación y aplique políticas de seguridad (límite de tasa, CORS, validación de esquemas, etc.).
- **Validación y saneamiento de entrada:** Cada endpoint valida estrictamente los JSON entrantes contra un esquema (OpenAPI) y rechaza entradas inválidas o malformadas. Esto previene ataques de inyección. También se aplican políticas OWASP (por ej. tamaño máximo de payload).
- **Roles y segregación de funciones:** Los datos sensibles (por ej. monedas alternas, descuentos estratégicos) solo pueden ser modificados por usuarios con privilegios adecuados. Los logs de auditoría incluyen quién realizó cada cambio.
- **Secretos y cifrado:** Las credenciales (claves JWT, conexiones a la BD) se almacenan en un vault seguro. La base de datos cifra datos sensibles en reposo si aplica.
- **Contenedores y runtime seguros:** Se debe endurecer la imagen del servicio (límites de recursos, usuario no root) y aplicar escaneo de vulnerabilidades. También se recomienda habilitar *contextos de seguridad* en el orquestador (e.g. Open Policy Agent).

En conjunto, se implementan autentificación/autorización robustas, TLS extremo a extremo, seguridad en contenedores y monitoreo centralizado, siguiendo las mejores prácticas de seguridad en microservicios. Por ejemplo, Atlassian recomienda cifrar todo el tráfico, usar tokens seguros y monitorear continuamente los servicios para detectar anomalías.

## Monitoreo y Métricas\_lp

Para mantener alta disponibilidad y rendimiento, **ListaPreciosBC** debe exponer métricas y logs detallados:

- **Métricas de desempeño:** Instrumentar el código (p.ej. con Micrometer) para recolectar métricas como latencia de respuestas, tasa de errores HTTP, recuento de solicitudes (por endpoint y por resultado) y uso de recursos (CPU, memoria). Registrar también métricas de dominio: número de listas creadas, precios aplicados, importaciones realizadas, etc. Estas métricas se exponen en formatos compatibles con Prometheus y se visualizan con dashboards en Grafana.
- **Monitoreo centralizado:** Como recomienda Atlassian, combinar logs estructurados y métricas proporciona una visión clara del sistema. Se debe usar Prometheus + Grafana (o equivalente) para métricas e infraestructura, y un stack de logs (ELK/EFK o Splunk) para almacenamiento de logs. Configurar alertas (por ejemplo, vía Alertmanager o Datadog) para umbrales críticos (latencia alta, tasa de error en aumento, fallos en importación masiva, etc.).
- **Health checks y trazas:** Implementar endpoints de *liveness* y *readiness* según estándares (por ej. `/actuator/health` en Spring Boot) para detección automática de fallos por el orquestador (k8s). También se recomienda habilitar trazas distribuidas (OpenTelemetry/Jaeger) para seguir la huella de una solicitud en el flujo (desde la petición de precio hasta la respuesta), facilitando la depuración de cuellos de botella.
- **Alertas de negocio:** Más allá de indicadores técnicos, monitorear eventos críticos: cuando se crea o expira una lista importante, enviar notificaciones internas. Por ejemplo, emitir métricas al sistema de indicadores con los precios aplicados en comprobantes, alimentando dashboards de margen de ganancia.

En resumen, se implementa un **sistema de observabilidad** completo: métricas en tiempo real (Prometheus/Grafana), logs estructurados y trazas distribuidas. Estas prácticas centralizadas de monitoreo ayudan a identificar problemas antes de que se conviertan en incidentes mayores

## Pruebas (Testing)\_lp

La estrategia de pruebas para ListaPreciosBC debe garantizar la calidad en cada nivel:

- **Pruebas unitarias:** Validar la lógica del dominio (entidades y políticas). Por ejemplo, tests de unidad para asegurarse que no se crean listas duplicadas (RN-LP-001), que los rangos de volumen y fecha son válidos (RN-LP-002, RN-LP-003), que `PrecioEspecifico.valor > 0`, etc. Usar frameworks como JUnit + Mockito (Java) o equivalentes.
- **Pruebas de integración:** Verificar la correcta interacción con la base de datos y otros componentes. Por ejemplo, testear que al crear una lista en BD se genera el historial correspondiente. Se pueden usar `testcontainers` o bases de datos en memoria para no depender de entornos externos. También incluir pruebas de integración con brokers de mensajería (simulando eventos entrantes y salientes).
- **Pruebas de API (contrato):** Realizar pruebas de integración contra los endpoints REST expuestos, usando herramientas como Postman/Newman o frameworks de pruebas integradas. Incluir casos típicos (flujo feliz) y alternativos (datos inválidos, autorización denegada). Para asegurar compatibilidad con otros bounded contexts, se recomienda **pruebas de contrato** (por ejemplo con Pact) sobre los eventos publicados/consumidos (p.ej. esquemas JSON de eventos `PrecioAplicado`, `ListaPrecioCreada`).
- **Pruebas end-to-end:** En un entorno que simule el sistema completo, generar escenarios de negocio: importar un CSV, emitir comprobantes para clientes concretos, asegurar que se aplican los precios correctos. Estas pruebas pueden automatizarse con scripts o herramientas de pruebas E2E.
- **Automatización y CI/CD:** Integrar todos los tests en el pipeline de CI/CD. Cada cambio de código activa la ejecución de pruebas unitarias e integrales. Como se destaca en buenas prácticas, esto permite detectar regresiones temprano y prevenir fallos en producción.
- **Cobertura y calidad:** Utilizar análisis de cobertura de código, linters y análisis estático (p.ej. SonarQube) para mantener estándares de calidad.
- **Otras pruebas:** Incorporar pruebas de carga/performance para validar escalabilidad (especialmente en operaciones de importación masiva) y, si es relevante, pruebas de seguridad (scan de dependencias, pruebas de penetración de API).

Siguiendo recomendaciones de la industria, el entorno de pruebas debe reflejar al máximo el ambiente de producción: datos realistas, servicios simulados si es necesario y contenedores replicando la infraestructura. En particular, integrar las pruebas en CI/CD garantiza calidad continua. Al automatizar todos los niveles de prueba (unitarias, integración, contrato, end-to-end) se refuerza la confiabilidad del servicio ListaPreciosBC.

# Decisiones de Arquitectura\_lp

Las decisiones arquitectónicas clave para **ListaPreciosBC** consideran patrones de microservicios y DDD:

- **Microservicio independiente:** ListaPreciosBC se implementa como un servicio autónomo con API REST. Sigue patrones de DDD: el agregado raíz es *ListaPrecio* y se diseñan entidades (*PrecioEspecifico*, *HistorialPrecio*) y objetos de valor para reflejar reglas de negocio. Gracias a esto, el servicio es *cohesivo* y encapsula toda la lógica tarifaria, cumpliendo el principio de responsabilidades claras. Como indica Microsoft Learn, utilizar entidades y agregados asegura microservicios «*loosely coupled and cohesive*».
- **Base de datos propia (Database per service):** Se emplea un esquema DB dedicado para ListaPreciosBC. Esto evita dependencia de esquemas externos y permite escoger el modelo de datos óptimo (relacional para consultas complejas de rangos, por ejemplo). Cada cambio solo afecta esta base, manteniendo bajo acoplamiento con otros servicios.
- **Comunicación asíncrona por eventos:** ListaPreciosBC se integra con otros contextos vía eventos de dominio. Publica eventos como `ListaPrecioCreada`, `PrecioEspecificoModificado` o `PrecioAplicado` cuando corresponda. De este modo, otros servicios (por ej. *IndicadoresNegocioBC* o *ComprobantesElectronicosBC*) reaccionan sin invocación directa. Para garantizar consistencia eventual en transacciones de varias etapas (por ej. al inhabilitar listas si no hay comprobantes activos), se podría emplear el patrón *Saga*. Asimismo, para consultas complejas que requieran datos de múltiples servicios, se pueden aplicar patrones como *API Composition* o *CQRS* manteniendo vistas materiales actualizadas por eventos.
- **API Gateway y contratos claros:** Se define un contrato REST/JSON claro, documentado con OpenAPI (Swagger). El API Gateway unifica puntos de entrada (comprobación de auth, enrutamiento) y podría implementar *Backends for Frontends* si hay múltiples UI. Esto simplifica la evolución de la API sin afectar a los clientes.
- **Infraestructura cloud-nativa:** Se planea desplegar en contenedores (Docker) orquestados (Kubernetes). Esto habilita autoescalado ante alta demanda (p.ej. en periodos de promociones). También permite políticas de reinicio y tolerancia a fallos. Se aplican *circuit breakers* o timeouts (por ej. con Resilience4j) para evitar cascadas.
- **Herramientas y frameworks:** Se escoge un stack industrial (por ej. Spring Boot o .NET Core) con soporte para inyección de dependencias y patrones DDD. Para mensajería se puede usar Kafka o RabbitMQ, asegurando durabilidad de eventos. El código sigue arquitecturas limpias (*clean architecture*) separando dominio, aplicación e infraestructura.
- **Decisiones de escalabilidad y rendimiento:** Se considera el *caché* de respuestas frecuentes (p.ej. listas base) para reducir latencias. Se definen índices y particiones en BD si se espera gran volumen de precios históricos. Las importaciones masivas usan procesamiento por lotes para no saturar el sistema.
- **Observabilidad y operabilidad:** Finalmente, se enfatiza la trazabilidad: logs con IDs de correlación, métricas expuestas y monitoreo continuo, cumpliendo la filosofía de microservicios observable. Estas decisiones arquitectónicas alinean ListaPreciosBC con buenas prácticas (base de datos por servicio, eventos para integración) y permiten evolucionar el sistema de precios de forma independiente.

En resumen, ListaPreciosBC se diseña como un servicio desacoplado, seguro y observable, con un modelo de datos rico en dominio, políticas de seguridad integrales, monitoreo activo y pruebas automatizadas. Estas secciones (11-15) completan la documentación mostrando cómo se implementa y mantiene esta capa crítica del dominio de precios, siguiendo patrones DDD y de microservicios comprobados