Tecnológico de Costa Rica

Curso:

Programación orientada a objetos

Documento:

Olores de código y deuda técnica

Estudiantes:

Diego Castillo Fallas

David Fernández Torres

Josimar Spencer Suarez

Profesor

Luis Javier Chavarría Sánchez

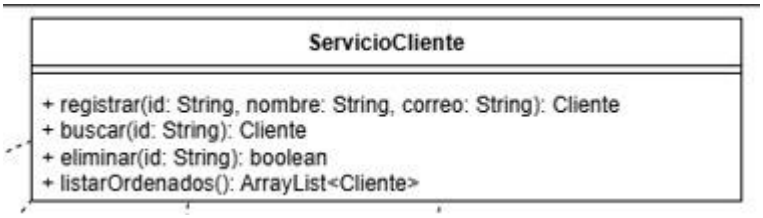
Semestre

2

Año

2025

```java
package Remontando.util;

import Remontando.model.Cliente;
import java.util.ArrayList;

/**
 * Servicio encargado de gestionar todas las operaciones relacionadas
 * con los clientes del sistema.
 * Las operaciones contempladas incluyen: registrar clientes, buscar
 * clientes, eliminar clientes y devolver una lista ordenada por nombre.</p>
 */
public class ServicioClientes {

    /**
     * Registra un nuevo cliente después de validar sus datos.
     * @param id identificación única del cliente
     * @param nombre nombre completo del cliente
     * @param correo correo electrónico del cliente
     * @return el cliente registrado
     * @throws IllegalArgumentException si los datos no son válidos o si el ID ya existe
     */
    public Cliente registrar(String id, String nombre, String correo) {

        // Validaciones de entrada
        Validador.texto(id, "ID");
        Validador.texto(nombre, "nombre");
        Validador.correo(correo);

        // Verificar duplicados
        if (buscar(id) != null) {
            throw new IllegalArgumentException("Ese ID ya existe");
        }

        // Generación del código del cliente
        String codigo = GeneradorCodigo.siguiente();

        // Crear y almacenar el cliente
        Cliente c = new Cliente(id, nombre, correo, codigo);
        Memoria.CLIENTES.add(c);

        return c;
    }

    /**
     * Busca un cliente en memoria por su ID.
     * @param id identificación del cliente
     * @return el cliente encontrado o null si no existe
     */
    public Cliente buscar(String id) {
        for (Cliente c : Memoria.CLIENTES) {
            if (c.getId().equals(id)) {
                return c;
            }
        }
        return null;
    }

    /**
     * Elimina un cliente del sistema, pero mantiene los instrumentos
     * asociados en la lista global (según requerimientos del proyecto).
     * @param id identificación del cliente a eliminar
     * @return true si el cliente fue eliminado, false si no existía
     */
    public boolean eliminar(String id) {

        Cliente c = buscar(id);

        if (c == null) {
            return false;
        }

        // Eliminarlo de la memoria principal
        return Memoria.CLIENTES.remove(c);
    }

    /**
     * Devuelve una lista ordenada alfabéticamente por nombre del cliente.
     * La lista retornada es una copia, protegiendo así la lista global en Memoria.
     * @return lista de clientes ordenados por nombre
     */
    public ArrayList<Cliente> listarOrdenados() {

        ArrayList<Cliente> copia = new ArrayList<>(Memoria.CLIENTES);
        Ordenamiento.ordenarClientes(copia);

        return copia;
    }
}
```

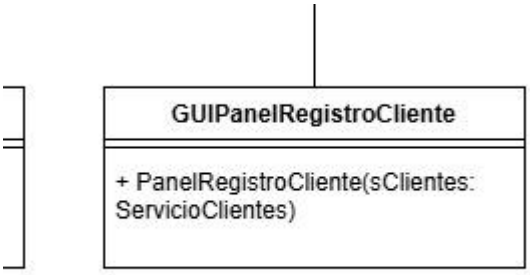


Opacidad:

En lugar de dejar que la UI lo haga todo, separamos el flujo en tres partes claras: la UI solo normaliza la entrada (convierte strings y muestra errores simples), el servicio toma las decisiones de negocio (qué crear, qué tasa aplicar) y un Formateador genera la salida para GUI/CLI. Así, cuando cambie una regla una tasa, un tipo nuevo o el formato del reporte solo

tocas la fábrica o el formateador. Menos puntos de ruptura, cambios más localizados y más fácil de probar.

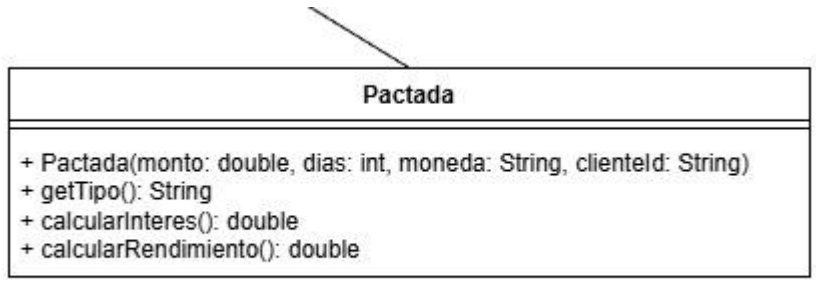
```java
package Remontando.view.GUI;

import javax.swing.*;
import java.awt.*;

import Remontando.util.ServicioClientes;
import Remontando.model.Cliente;

/**
 * Panel gráfico encargado de registrar un nuevo cliente en el sistema.
 * Este panel pertenece a la capa de presentación (GUI) y se limita a
 * interactuar con el usuario, delegando toda la lógica de negocio al
 * servicio ServicioClientes, en cumplimiento con el principio de
 * separación de responsabilidades (SoC).
 *
 * <p>El panel contiene un pequeño formulario con campos para ID, nombre
 * y correo electrónico, además de un botón que ejecuta el registro.
 */
public class PanelRegistroCliente extends JPanel {

    /**
     * Construye el panel que permite registrar clientes.
     * @param sClientes servicio encargado de gestionar los clientes
     */
    public PanelRegistroCliente(ServicioClientes sClientes) {

        setLayout(new BorderLayout());

        JPanel form = new JPanel(new GridLayout(4, 2, 5, 5));

        JTextField txtId = new JTextField();
        JTextField txtNombre = new JTextField();
        JTextField txtCorreo = new JTextField();

        form.add(new JLabel("ID:"));
        form.add(txtId);

        form.add(new JLabel("Nombre completo:"));
        form.add(txtNombre);

        form.add(new JLabel("Correo electrónico:"));
        form.add(txtCorreo);

        JButton btn = new JButton("Registrar Cliente");
        form.add(btn);

        form.add(new JLabel("ID:"));
        form.add(txtId);

        form.add(new JLabel("Nombre completo:"));
        form.add(txtNombre);

        form.add(new JLabel("Correo electrónico:"));
        form.add(txtCorreo);

        JButton btn = new JButton("Registrar Cliente");
        form.add(btn);

        add(form, BorderLayout.NORTH);

        JTextArea salida = new JTextArea();
        salida.setEditable(false);
        salida.setFont(new Font("Monospaced", Font.PLAIN, 13));

        JScrollPane scroll = new JScrollPane(salida);
        add(scroll, BorderLayout.CENTER);

        btn.addActionListener(e -> {
            try {

                // Registrar el cliente (lógica delegada al servicio)
                Cliente c = sClientes.registrar(
                        txtId.getText(),
                        txtNombre.getText(),
                        txtCorreo.getText()
                );

                // Construcción de salida formateada
                StringBuilder sb = new StringBuilder();
                sb.append("--- Cliente registrado exitosamente ---\n");
                sb.append("Nombre:  ").append(c.getNombre()).append("\n");
                sb.append("ID:      ").append(c.getId()).append("\n");
                sb.append("Correo:  ").append(c.getCorreo()).append("\n");
                sb.append("Código asignado: ").append(c.getCodigo()).append("\n");
                sb.append("----------------------------------------\n");

                salida.setText(sb.toString());

            } catch (Exception ex) {
                JOptionPane.showMessageDialog(this, ex.getMessage());
            }
        });
```



Viscosidad:

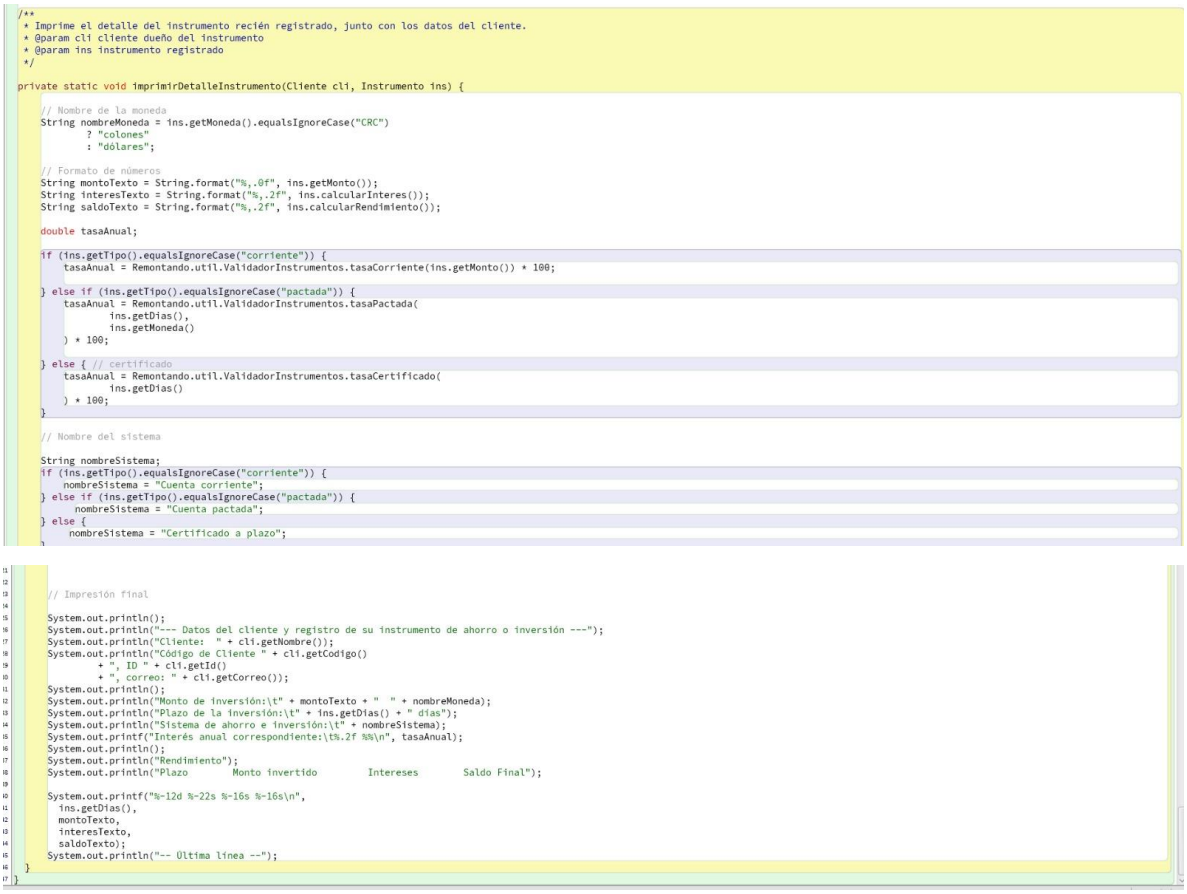
Con el panel simplemente recoge campos, los normaliza mínimamente y llama a ServicioClientes para registrar. No crea clientes directamente ni manipula colecciones compartidas. Eso hace que la “vía oficial” usar ServicioClientes sea la más obvia y la más fácil de usar. El flujo muestra que se apoya en Validador antes de pasar al servicio. Cuando la validación está en un solo sitio, los desarrolladores no sienten la tentación de re implementar reglas en 3 lugares distintos. Los controles y el flujo están organizados. Un lector entiende rápido “rellena, validar, llamar servicio mostrar resultado”, por lo que replicar el comportamiento en otro lugar sería redundante y menos atractivo que reusar este panel/servicio.

```java
package Remontando.model;

import Remontando.util.ValidadorInstrumentos;

/**
 * Representa una inversión a la vista pactada.
 * Permite CRC y USD según reglas del negocio.
 */
public class Pactada extends Instrumento {

    /**
     * Crea una inversión pactada.
     * @param monto monto invertido
     * @param dias plazo total
     * @param moneda CRC
     * @param clienteId código del cliente
     */
    public Pactada(double monto, int dias, String moneda, String clienteId) {
        super(monto, dias, moneda, clienteId);
        ValidadorInstrumentos.validarPactada(monto, dias, moneda);
    }

    @Override
    public String getTipo() {
        return "pactada";
    }

    @Override
    public double calcularInteres() {
        double tasa = ValidadorInstrumentos.tasaPactada(dias, moneda);
        return monto * tasa * (dias / 30.0);
    }

    @Override
    public double calcularRendimiento() {
        return monto + calcularInteres();
    }
}
```

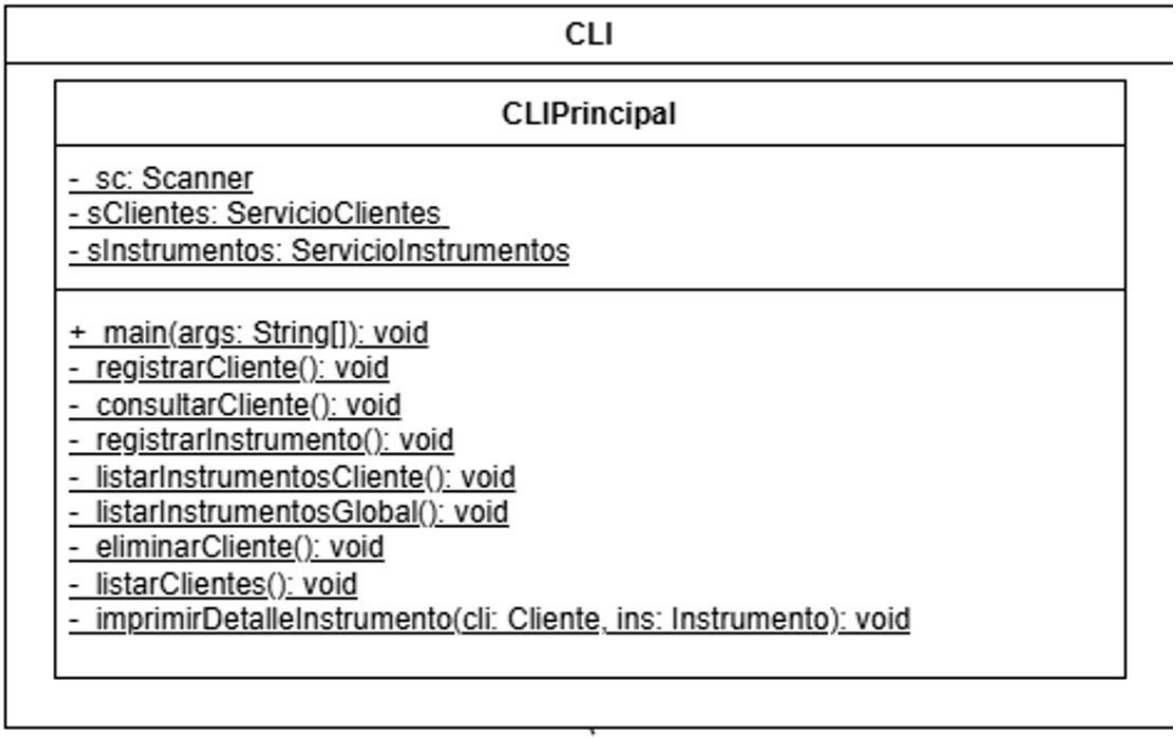


Complejidad innecesaria:

Lo evita al separar cada tipo de instrumento en su propia subclase (Corriente / Pactada / Certificado) hace que el comportamiento específico esté exactamente donde esperamos encontrarlo. Cuando tienes que leer o cambiar la fórmula de cálculo, la tasa aplicada o el nombre del producto, vas directo a la clase correspondiente y no a un switch disperso en la UI o en un servicio. Esto reduce la sensación y facilita el mantenimiento diario: añadir o corregir una regla es menos costoso porque la responsabilidad está localizada. A mayor claridad para nuevos desarrolladores, menos riesgo de romper otras cosas al tocar una clase concreta, y pruebas unitarias más simples (puedes testear cada subclase en aislamiento). Además, favorece la extensión controlada, si mañana aparece un nuevo tipo, añades una subclase con su comportamiento y listo en vez de parchear condicionales por todos lados.



```java
/**
 * Imprime el detalle del instrumento recién registrado, junto con los datos del cliente.
 * @param cli cliente dueño del instrumento
 * @param ins instrumento registrado
 */
private static void imprimirDetalleInstrumento(Cliente cli, Instrumento ins) {

    // Nombre de la moneda
    String nombreMoneda = ins.getMoneda().equalsIgnoreCase("CRC")
            ? "colones"
            : "dólares";

    // Formato de números
    String montoTexto = String.format("%,.0f", ins.getMonto());
    String interesTexto = String.format("%,.2f", ins.calcularInteres());
    String saldoTexto = String.format("%,.2f", ins.calcularRendimiento());

    double tasaAnual;

    if (ins.getTipo().equalsIgnoreCase("corriente")) {
        tasaAnual = Remontando.util.ValidadorInstrumentos.tasaCorriente(ins.getMonto()) * 100;

    } else if (ins.getTipo().equalsIgnoreCase("pactada")) {
        tasaAnual = Remontando.util.ValidadorInstrumentos.tasaPactada(
                ins.getDias(),
                ins.getMoneda()
        ) * 100;

    } else { // certificado
        tasaAnual = Remontando.util.ValidadorInstrumentos.tasaCertificado(
                ins.getDias()
        ) * 100;
    }

    // Nombre del sistema

    String nombreSistema;
    if (ins.getTipo().equalsIgnoreCase("corriente")) {
        nombreSistema = "Cuenta corriente";
    } else if (ins.getTipo().equalsIgnoreCase("pactada")) {
        nombreSistema = "Cuenta pactada";
    } else {
        nombreSistema = "Certificado a plazo";
    }

    // Impresión final

    System.out.println();
    System.out.println("--- Datos del cliente y registro de su instrumento de ahorro o inversión ---");
    System.out.println("Cliente:  " + cli.getNombre());
    System.out.println("Código de Cliente " + cli.getCodigo()
            + ", ID " + cli.getId()
            + ", correo: " + cli.getCorreo());
    System.out.println();
    System.out.println("Monto de inversión:\t" + montoTexto + "  " + nombreMoneda);
    System.out.println("Plazo de la inversión:\t" + ins.getDias() + " días");
    System.out.println("Sistema de ahorro e inversión:\t" + nombreSistema);
    System.out.printf("Interés anual correspondiente:\t%.2f %%\n", tasaAnual);
    System.out.println();
    System.out.println("Rendimiento");
    System.out.println("Plazo        Monto invertido        Intereses       Saldo Final");

    System.out.printf("%-12d %-22s %-16s %-16s\n",
      ins.getDias(),
      montoTexto,
      interesTexto,
      saldoTexto);
    System.out.println("-- Última línea --");
  }
}
```

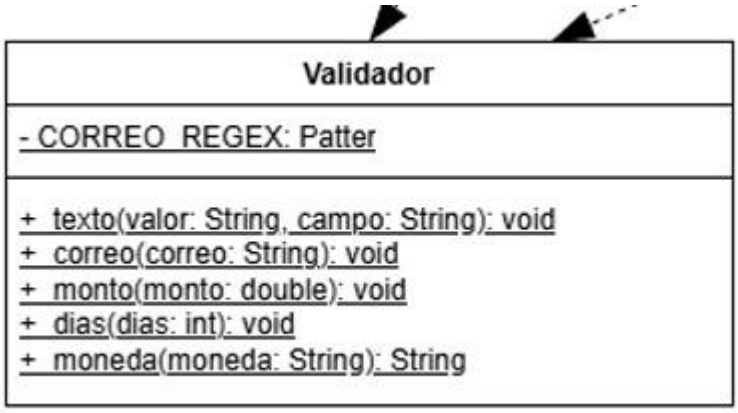


Inmovilidad:

En vez de depender de campos concretos de Instrumento o Cliente en cada vista, la salida se construye en un punto lógico para que la representación (texto/formatos) no esté repetidamente ligada a la forma concreta del modelo. Normalizar tipos/valores lo antes posible (por ejemplo: monedas y tipos de instrumento no tratados como cadenas libres en toda la app, sino representados por un pequeño adaptador) para que cambios en el dominio no obliguen a tocar las vistas. Proveer un único "punto de verdad" para la representación textual. Con eso, el resto del código llama a ese y no necesita saber cómo está guardado el Instrumento.

```java
*/
public class Validador {

    /** Expresión regular que valida el formato estándar de un correo electrónico. */
    private static final Pattern CORREO_REGEX =
            Pattern.compile("^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$");

    /**
     * Valida que un texto no sea nulo ni vacío.
     * @param valor texto a verificar
     * @param campo nombre del campo (para mostrar en el error)
     * @throws IllegalArgumentException si el texto es nulo o vacío
     */
    public static void texto(String valor, String campo) {
        if (valor == null || valor.trim().isEmpty()) {
            throw new IllegalArgumentException("Debe ingresar " + campo);
        }
    }

    /**
     * Valida que el correo electrónico tenga un formato correcto.
     * @param correo correo a validar
     * @throws IllegalArgumentException si el formato no es válido
     */
    public static void correo(String correo) {
        if (correo == null || !CORREO_REGEX.matcher(correo).matches()) {
            throw new IllegalArgumentException("Correo inválido");
        }
    }

    /**
     * Valida que el monto sea mayor a cero.
     * @param monto valor del monto
     * @throws IllegalArgumentException si el monto es menor o igual a cero
     */
    public static void monto(double monto) {
        if (monto <= 0) {
            throw new IllegalArgumentException("Monto inválido");
        }
    }
```

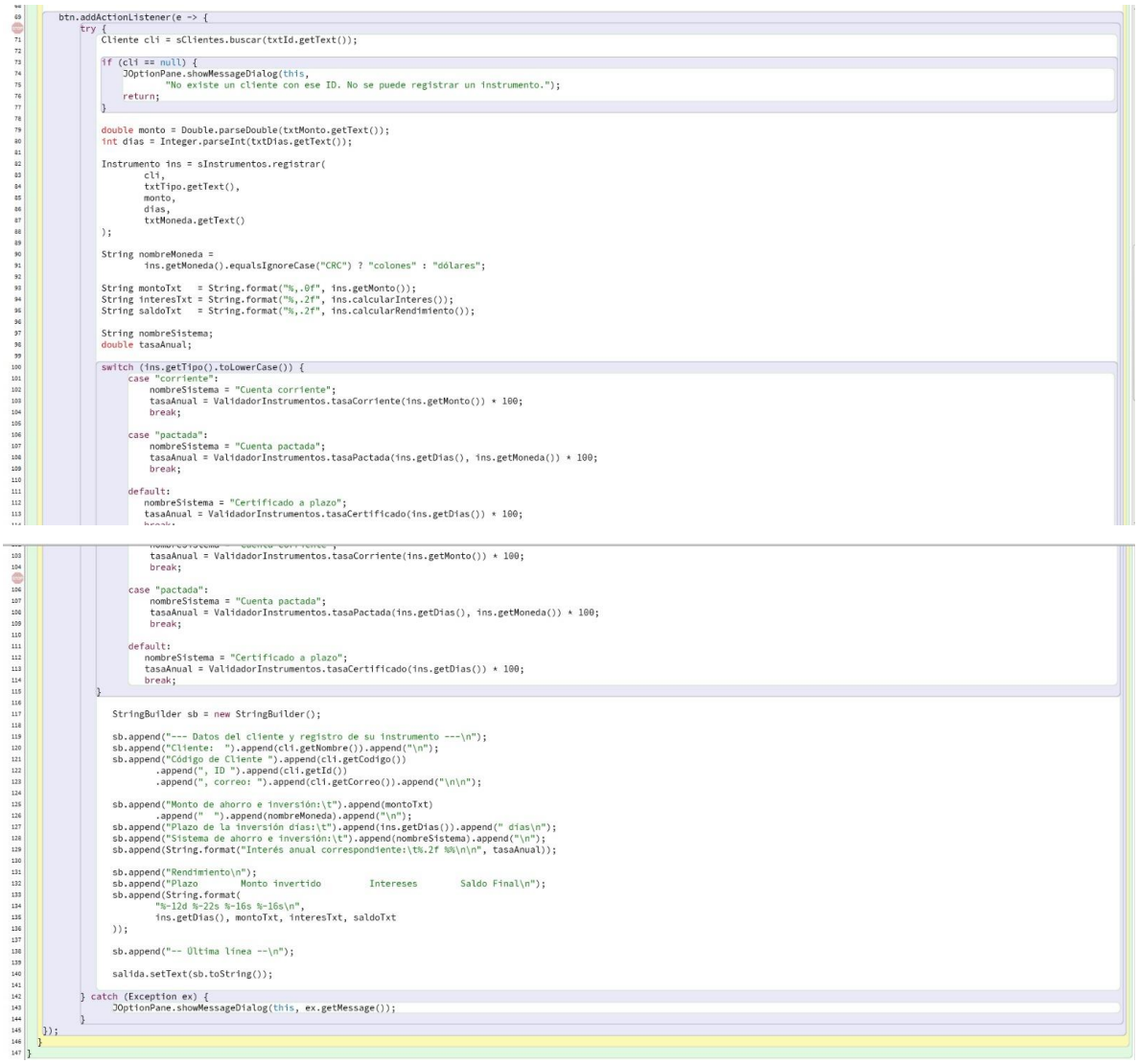
```java
    /**
     * Valida que la cantidad de días sea mayor a cero.
     * @param dias cantidad de días
     * @throws IllegalArgumentException si los días son menores o iguales a cero
     */
    public static void dias(int dias) {
        if (dias <= 0) {
            throw new IllegalArgumentException("Días inválidos");
        }
    }

    /**
     * Valida la moneda ingresada y la normaliza a mayúsculas.
     * Solo se aceptan CRC y USD.
     * @param moneda texto ingresado por el usuario
     * @return moneda en formato normalizado (CRC o USD)
     * @throws IllegalArgumentException si la moneda no es válida
     */
    public static String moneda(String moneda) {

        if (moneda == null || moneda.trim().isEmpty()) {
            throw new IllegalArgumentException("Moneda inválida");
        }

        String m = moneda.trim().toUpperCase();

        if (!m.equals("CRC") && !m.equals("USD")) {
            throw new IllegalArgumentException("Moneda inválida");
        }

        return m;
    }
}
```



Rigidez:

Centralizar las validaciones en una única clase reutilizable y con métodos bien nombrados en lugar de repetir comprobaciones por todo el código. Esa centralización hace que las reglas de validación (y sus cambios) queden en un solo punto en vez de dispersas en servicios, vistas. Normalizar entradas en ese punto frontal por ejemplo moneda, comprobación por valores permitidos para que el resto del sistema reciba valores ya saneados y no tenga que

repetir lógica de normalización. Usar excepciones claras desde el validador para que la política de fallo sea consistente y el manejo de errores quede homogéneo en las capas superiores.

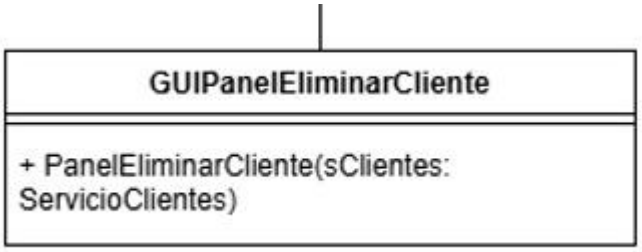


```
    btn.addActionListener(e -> {
        try {
            Cliente cli = sClientes.buscar(txtId.getText());

            if (cli == null) {
                JOptionPane.showMessageDialog(this,
                        "No existe un cliente con ese ID. No se puede registrar un instrumento.");
                return;
            }

            double monto = Double.parseDouble(txtMonto.getText());
            int dias = Integer.parseInt(txtDias.getText());

            Instrumento ins = sInstrumentos.registrar(
                    cli,
                    txtTipo.getText(),
                    monto,
                    dias,
                    txtMoneda.getText()
            );

            String nombreMoneda =
                    ins.getMoneda().equalsIgnoreCase("CRC") ? "colones" : "dólares";

            String montoTxt   = String.format("%,.0f", ins.getMonto());
            String interesTxt = String.format("%,.2f", ins.calcularInteres());
            String saldoTxt   = String.format("%,.2f", ins.calcularRendimiento());

            String nombreSistema;
            double tasaAnual;

            switch (ins.getTipo().toLowerCase()) {
                case "corriente":
                    nombreSistema = "Cuenta corriente";
                    tasaAnual = ValidadorInstrumentos.tasaCorriente(ins.getMonto()) * 100;
                    break;

                case "pactada":
                    nombreSistema = "Cuenta pactada";
                    tasaAnual = ValidadorInstrumentos.tasaPactada(ins.getDias(), ins.getMoneda()) * 100;
                    break;

                default:
                    nombreSistema = "Certificado a plazo";
                    tasaAnual = ValidadorInstrumentos.tasaCertificado(ins.getDias()) * 100;
                    break;
            }

            StringBuilder sb = new StringBuilder();

            sb.append("--- Datos del cliente y registro de su instrumento ---\n");
            sb.append("Cliente:  ").append(cli.getNombre()).append("\n");
            sb.append("Código de Cliente ").append(cli.getCodigo())
                    .append(", ID ").append(cli.getId())
                    .append(", correo: ").append(cli.getCorreo()).append("\n\n");

            sb.append("Monto de ahorro e inversión:\t").append(montoTxt)
                    .append("  ").append(nombreMoneda).append("\n");
            sb.append("Plazo de la inversión días:\t").append(ins.getDias()).append(" días\n");
            sb.append("Sistema de ahorro e inversión:\t").append(nombreSistema).append("\n");
            sb.append(String.format("Interés anual correspondiente:\t%.2f %%\n\n", tasaAnual));

            sb.append("Rendimiento\n");
            sb.append("Plazo        Monto invertido         Intereses        Saldo Final\n");
            sb.append(String.format(
                    "%-12d %-22s %-16s %-16s\n",
                    ins.getDias(), montoTxt, interesTxt, saldoTxt
            ));

            sb.append("-- Última línea --\n");

            salida.setText(sb.toString());

        } catch (Exception ex) {
            JOptionPane.showMessageDialog(this, ex.getMessage());
        }
    });
  }
}
```

Fragilidad:

Centralizar responsabilidades y normalizar los puntos de cambio: en vez de permitir que la UI haga validaciones, decisiones por tipo y formateo todo a la vez, reorganizamos el flujo en tres piezas bien definidas. Primero, en el borde (la UI) solo limpiamos y convertimos la entrada: transformamos strings en tipos y rechazamos lo que no se puede convertir. Segundo, en el núcleo ponemos la orquestación y las reglas de negocio aquí se decide qué objeto crear, qué tasa aplicar y qué reglas de validación de negocio deben cumplirse. Tercero, la presentación queda en un Formateador reutilizable que genera la cadena o la tabla de salida para GUI y CLI. Con esto, cuando cambie una tasa, el formato del reporte o el backend, solo hay que tocar la fábrica o el formateador: no hace falta buscar y actualizar paneles. Es una forma práctica de reducir riesgos, facilitar pruebas y mantener el código legible.

```
 */
public class GUIPanelEliminarCliente extends JPanel {

    /**
     * Construye el panel para eliminar un cliente, mostrando el campo
     * para ingresar el ID y el botón para ejecutar la acción.
     * @param sClientes servicio encargado de gestionar a los clientes
     */
    public GUIPanelEliminarCliente(ServicioClientes sClientes) {

        setLayout(new BorderLayout());

        JPanel form = new JPanel(new GridLayout(1, 2, 5, 5));

        JTextField txtId = new JTextField();

        form.add(new JLabel("ID del cliente:"));
        form.add(txtId);

        add(form, BorderLayout.NORTH);

        JTextArea salida = new JTextArea();
        salida.setEditable(false);
        add(new JScrollPane(salida), BorderLayout.CENTER);

        JButton btn = new JButton("Eliminar");
        add(btn, BorderLayout.SOUTH);

        btn.addActionListener(e -> {

            boolean eliminado = sClientes.eliminar(txtId.getText());

            if (eliminado) {
                salida.setText("Cliente eliminado.\n");
            } else {
                salida.setText("No existe ese cliente.\n");
            }
        });
    }
}
```



Repetición innecesaria

En lugar de repetir lógica y mensajes en varias partes de la UI, se delegó la operación real en el servicio Clientes.eliminar y la vista simplemente consume el resultado y muestra un mensaje. Es decir: la vista no implementa la eliminación ni la lógica de negocio, solo orquesta la llamada y presenta el resultado. Además, la presentación del resultado está concentrada en un único sitio salida.setText, evitando copias del mismo texto en otros componentes

Listado para evitar la deuda técnica:

• Separar qué decide, qué procesa y qué muestra:

No mezclar responsabilidades; cada parte del sistema debe tener un rol claro.

• Poner las reglas en un solo lugar:

Toda regla de negocio o validación debe existir una sola vez y ser reutilizada.

• Evitar repetir lógica:

Cualquier comportamiento usado en más de un lugar debe convertirse en función o módulo común.

• Trabajar con datos limpios desde el inicio:

Convertir y normalizar entradas apenas ingresan al sistema para evitar errores posteriores.

• Diseñar para extender, no para modificar:

Agregar nuevas funciones mediante componentes nuevos, no parcheando código existente.

• Evitar que la interfaz tome decisiones:

La UI solo muestra y recoge datos; la lógica y las decisiones van en la capa de negocio.

• Mantener el flujo de procesamiento simple y predecible:

Entrada, validación, proceso y salida. Nada extra, nada mezclado.

• Tener variables significativas:

Cada variable o idea que se manipula debe ser clara y significativa para evitar gasto de tiempo innecesario