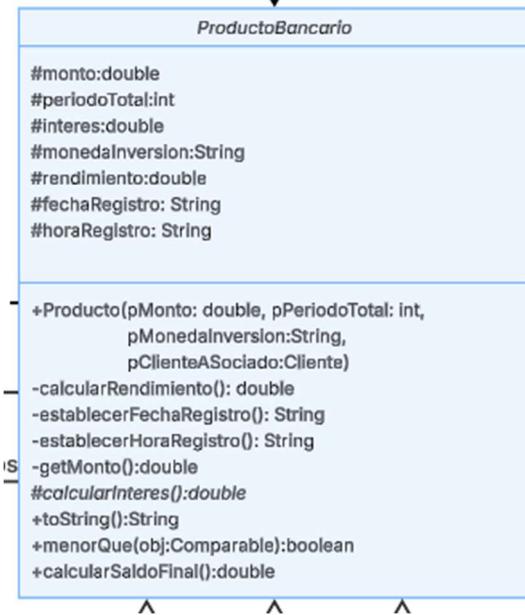


Opacidad



Para evitar el olor de “opacidad”, se evitó usar nombres poco representativos que pudieran generar interpretaciones erróneas con respecto a lo que hace un método, clase, atributo o variable, ejemplo de ello, se muestra la clase abstracta *ProductoBancario*, la cual cuenta con nombres representativos.

Repetición innecesaria

```
protected double calcularInteres() {  
    if (periodoTotal < 15) {  
        return 0;  
    } else if (periodoTotal < 30) {  
        return monedaInversion == "colones" ? 0.0485 : 0.008;  
    } else if (periodoTotal < 60) {  
        return monedaInversion == "colones" ? 0.0494 : 0.0091;  
    } else if (periodoTotal < 90) {  
        return monedaInversion == "colones" ? 0.0523 : 0.0106;  
    } else if (periodoTotal < 180) {  
        return monedaInversion == "colones" ? 0.0581 : 0.0144;  
    } else if (periodoTotal < 270) {  
        return monedaInversion == "colones" ? 0.0883 : 0.0221;  
    } else if (periodoTotal < 360) {  
        return monedaInversion == "colones" ? 0.0869 : 0.0226;  
    } else {  
        return monedaInversion == "colones" ? 0.0869 : 0.024;  
    }  
}
```

El mismo nombre define este olor, en este caso, se evitó este al simplificar que después de los 360 días el interés es el mismo, ignorando los demás rangos que establecía el contexto.

Complejidad innecesaria

```
protected double calcularInteres() {  
    if (periodoTotal < 15) {  
        return 0;  
    } else if (periodoTotal < 30) {  
        return monedaInversion == "colones" ? 0.0485 : 0.008;  
    } else if (periodoTotal < 60) {  
        return monedaInversion == "colones" ? 0.0494 : 0.0091;  
    } else if (periodoTotal < 90) {  
        return monedaInversion == "colones" ? 0.0523 : 0.0106;  
    } else if (periodoTotal < 180) {  
        return monedaInversion == "colones" ? 0.0581 : 0.0144;  
    } else if (periodoTotal < 270) {  
        return monedaInversion == "colones" ? 0.0883 : 0.0221;  
    } else if (periodoTotal < 360) {  
        return monedaInversion == "colones" ? 0.0869 : 0.0226;  
    } else {  
        return monedaInversion == "colones" ? 0.0869 : 0.024;  
    }  
}
```

El olor de complejidad innecesaria aparece cuando se hace más complicado el código, un ejemplo de esto se presenta con los condicionales, donde se puede reducir operadores ternarios en su lugar, en este caso, se usaron para definir el interés según la moneda. Esto permitió reducir lo que se haría en unas 4 o 5 líneas de código en una sola.

Inmovilidad

```
/**  
 * Valida si num1 < num2.  
 */  
public static boolean validarMenor(double num1, double num2) {  
    return num1 < num2;  
}
```

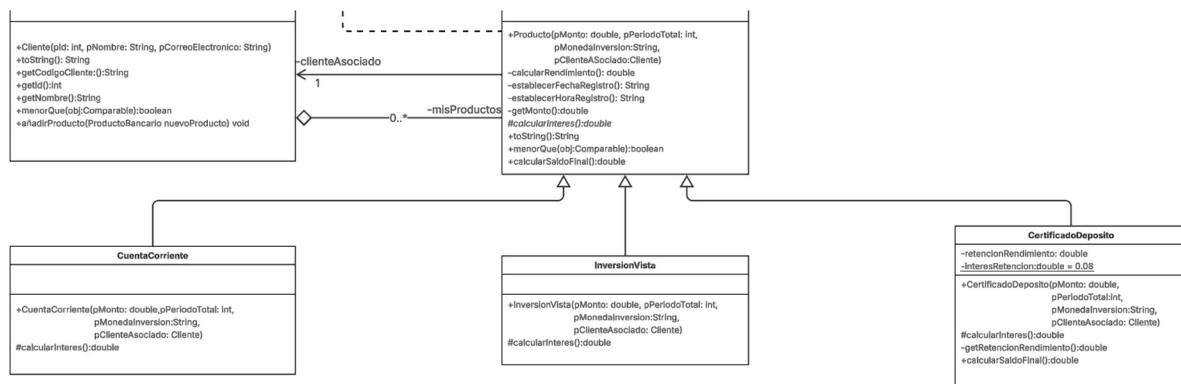
Este olor se presenta cuando es imposible reusar código, para evitar este problema, se aplicó el principio de divide y vencerás, creando pequeñas funciones para que juntas realicen una tarea específica y de ser necesario, poder reutilizar el código. En el ejemplo dado, se creó una función que valida el menor entre dos números, logrando reutilizarse en varias funciones de ser necesario.

Viscosidad

```
/**  
 * Validación de cuenta corriente.  
 * Mínimo: €25,000  
 */  
public static boolean validarCuentaCorriente(double monto, String moneda) {  
    if (!moneda.equalsIgnoreCase("colones")) {  
        return false;  
    }  
    return validarMenor(25000, monto);  
}
```

La viscosidad ocurre cuando hacer lo correcto es más difícil que no hacerlo, en este caso, habría sido más sencillo hacer “return 25000 < monto”, pero habría sido menos elegante y habría traído el olor de viscosidad, para evitar esto, se crea la función validarMenor que permite eliminar este olor.

Rigidez



La rigidez ocurre cuando un pequeño cambio causa una cascada de cambios necesarios, un ejemplo donde se evitó esto, es en la herencia de la clase ProductoBancario, si no se heredaran métodos comunes, cada clase tendría el mismo fragmento de código al menos en su mayoría (también implica repetición innecesaria), si se requiere cambiar un método común, habría que ir a todas las clases a cambiar el mismo fragmento de código.

Fragilidad

```
/**  
 * Valida monedas permitidas: colones o dólares.  
 */  
public static boolean validarMoneda(String moneda) {  
    if (moneda == null) {  
        return false;  
    }  
    moneda = moneda.toLowerCase();  
    return moneda.equals("colones") || moneda.equals("dolares");  
}
```

La fragilidad ocurre cuando algo puede romperse por un mal manejo del código, supongamos que el fragmento de código mostrado no se implementa, ¿qué pasaría si ingresa como valor de moneda “euros” ?, posiblemente una función que espere que la moneda sea dolares o colones entraría en estado de error, provocando que el programa caiga y sea frágil, para evitarlo, se crearon validaciones que eviten un código frágil.