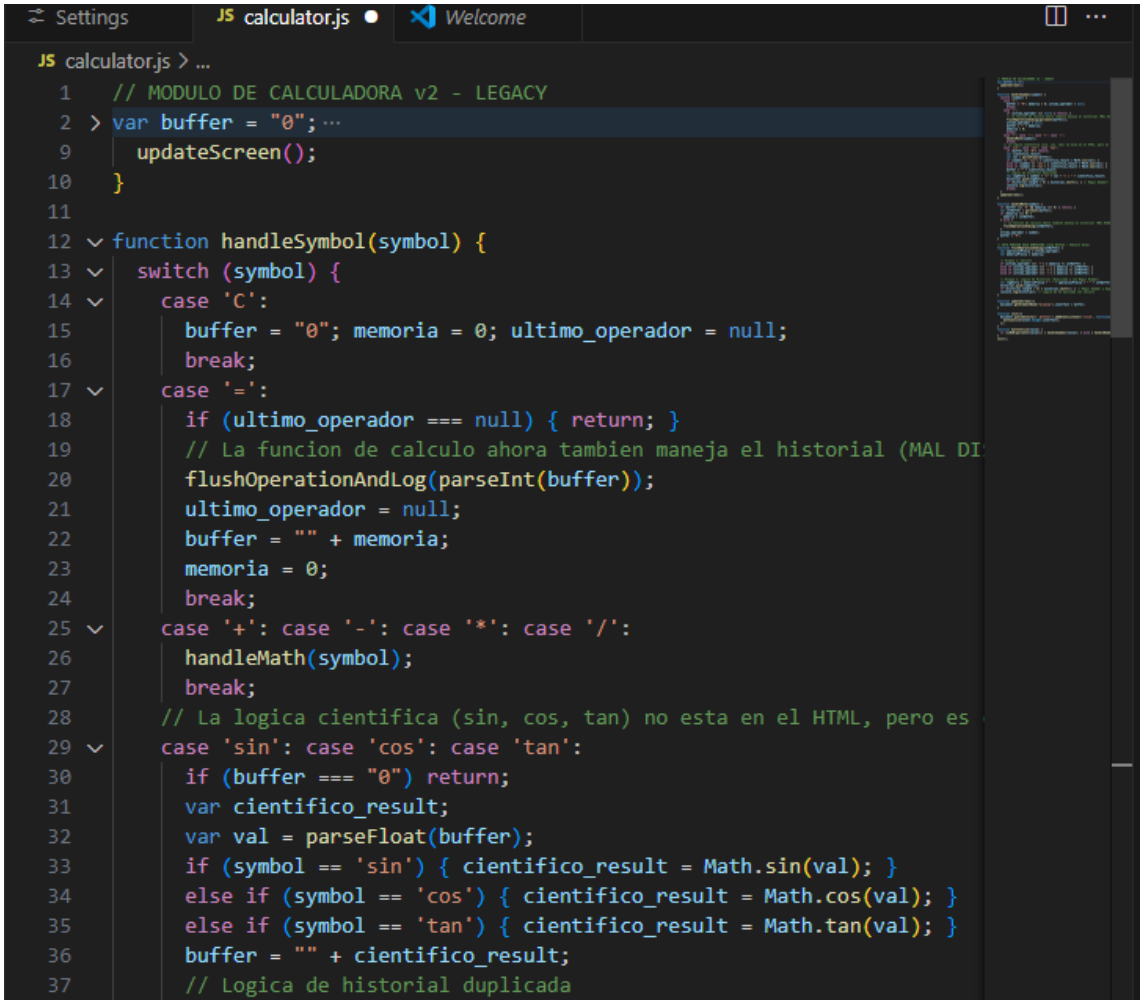
Taller de Refactorización y Métricas de Calidad

Fecha: 28 de October de 2025

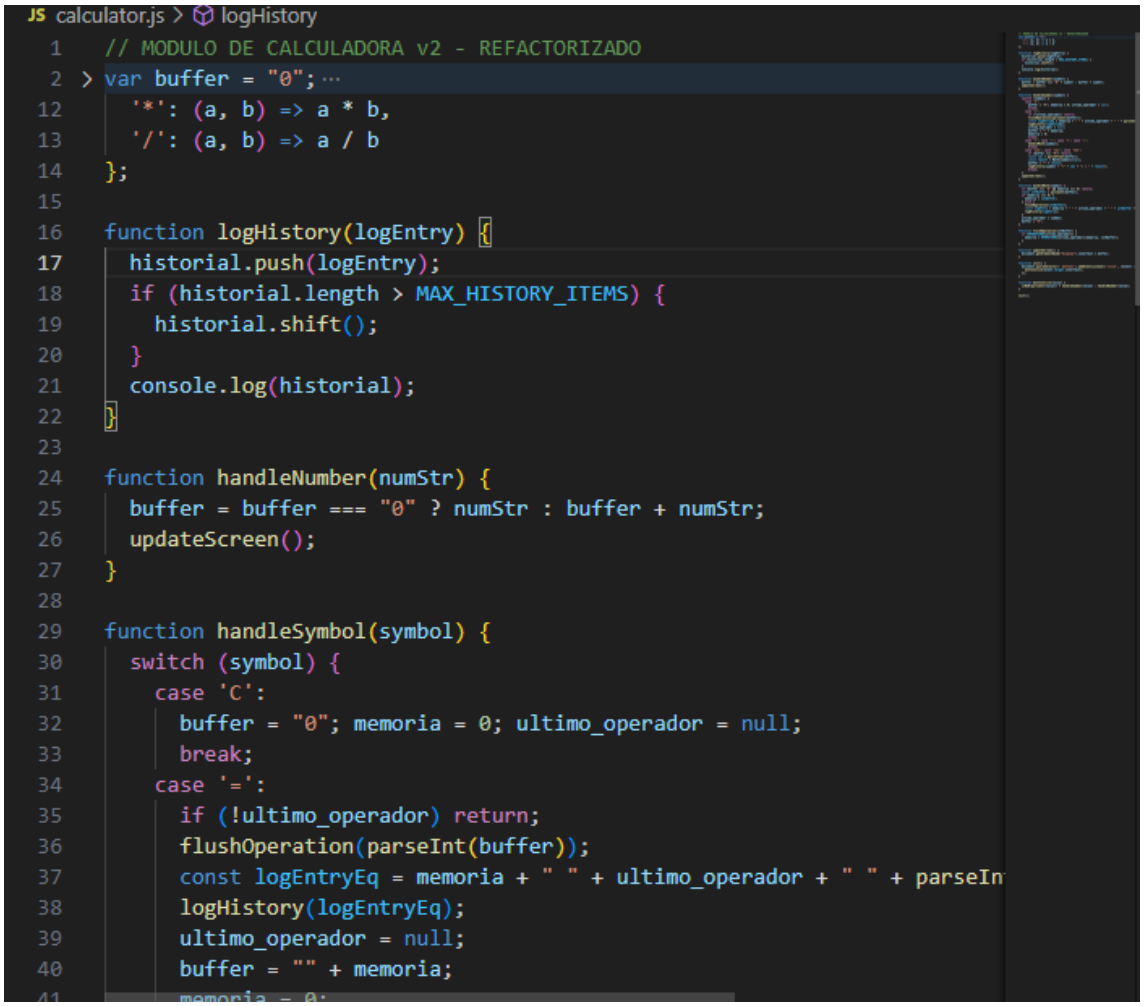
Nombre: Melanny Guate

1. **Auditoría Inicial / Initial Audit**

Antes de aplicar refactorización, el código legacy de la calculadora presentaba alta complejidad ciclomática, especialmente en las funciones handleSymbol (Complejidad ≈ 10) y flushOperationAndLog (Complejidad ≈ 12). Esto indicaba múltiples responsabilidades, condicionales anidados y duplicación de lógica.



Before refactoring, the legacy calculator code showed high cyclomatic complexity, particularly in the functions handleSymbol (Complexity ≈ 10) and flushOperationAndLog (Complexity ≈ 12). This revealed multiple responsibilities, nested conditionals, and duplicated logic.



2. Análisis de Code Smells / Code Smells Analysis

Duplicated Code (Violación de DRY):

El bloque de historial dentro de handleSymbol y flushOperationAndLog era idéntico. Example:

historial.push(logEntry);

if (historial.length > 5) { historial.shift(); } // Magic Number console.log(historial);

Magic Number: El número 5 aparece sin contexto en dos lugares del código.

Long Method: La función flushOperationAndLog realizaba cálculo y gestión de historial, violando el principio de responsabilidad única.

English:

The history block was duplicated in both handleSymbol and flushOperationAndLog. The number 5 was a magic number without explanation, and flushOperationAndLog performed both arithmetic and logging, violating the Single Responsibility Principle.

3. Proceso de Refactorización / Refactoring Process

Se aplicaron tres patrones principales de refactorización: Extract Constant, Extract Method y Replace Conditional with Strategy.

1. **Extract Constant:**

```
const MAX_HISTORY_ITEMS = 5
```

1. **Extract Method:**

```
function logHistory(logEntry) { historial.push(logEntry);
if (historial.length > MAX_HISTORY_ITEMS) { historial.shift(); } console.log(historial);
}
```

2. Replace Conditional with Strategy:

```
const OPERATIONS = { '+': (a,b)=>a+b, '-':(a,b)=>a-b, '*':(a,b)=>a*b, '/':(a,b)=>a/b };
function flushOperation(intBuffer){ if(OPERATIONS[ultimo_operador]) memoria
= OPERATIONS[ultimo_operador](memoria,intBuffer); }
```

English:

Applied Extract Constant, Extract Method, and Replace Conditional with Strategy patterns to reduce duplication, improve readability, and isolate responsibilities.

2. Auditoría Final / Final Audit

Después de la refactorización, las funciones principales redujeron drásticamente su complejidad ciclomática: flushOperation (Complejidad ≈ 2) y handleSymbol (Complejidad ≈ 4). El código es más legible, mantenible y fácil de probar.

Conclusión / Conclusion:

La reducción en la complejidad y eliminación de duplicaciones demuestra un código más limpio y alineado con los principios SOLID. This shows a cleaner, maintainable, and professional-grade software design.

CODIGO

```
// MODULO DE CALCULADORA v2 - REFACTORIZADO

var buffer = "0";

var memoria = 0;

var ultimo_operador;

var historial = [];


const MAX_HISTORY_ITEMS = 5;


const OPERATIONS = {
  '+': (a, b) => a + b,
  '-': (a, b) => a - b,
  '*': (a, b) => a * b,
  '/': (a, b) => a / b
};


function logHistory(logEntry) {
  historial.push(logEntry);
  if (historial.length > MAX_HISTORY_ITEMS) {
    historial.shift();
  }
  console.log(historial);
}
```

```
function handleNumber(numStr) {
  buffer = buffer === "0" ? numStr : buffer + numStr;
  updateScreen();
}

function handleSymbol(symbol) {
  switch (symbol) {
    case 'C':
      buffer = "0"; memoria = 0; ultimo_operador = null;
      break;
    case '=':
      if (!ultimo_operador) return;
      flushOperation(parseInt(buffer));
      const logEntryEq = memoria + " " + ultimo_operador + " " + parseInt(buffer) + " = " +
memoria;
      logHistory(logEntryEq);
      ultimo_operador = null;
      buffer = "" + memoria;
      memoria = 0;
      break;
    case '+': case '-': case '*': case '/':
      handleMath(symbol);
      break;
    case 'sin': case 'cos': case 'tan':
      if (buffer === "0") return;
      const val = parseFloat(buffer);
      const result = Math[symbol](val);
```

```
      buffer = "" + result;
      logHistory(symbol + "(" + val + ") = " + result);
      break;
    }
    updateScreen();
}

function handleMath(symbol) {
    if (buffer === '0' && memoria === 0) return;
    const intBuffer = parseInt(buffer);
    if (memoria === 0) {
      memoria = intBuffer;
    } else {
      flushOperation(intBuffer);
      const logEntry = memoria + " " + ultimo_operador + " " + intBuffer + " = " + memoria;
      logHistory(logEntry);
    }
    ultimo_operador = symbol;
    buffer = "0";
}

function flushOperation(intBuffer) {
    if (OPERATIONS[ultimo_operador]) {
      memoria = OPERATIONS[ultimo_operador](memoria, intBuffer);
    }
}

function updateScreen() {
```

```
  document.getElementById("display").innerText = buffer;
}

function init() {
  document.querySelector('.buttons').addEventListener('click', (event) => {
    buttonClick(event.target.innerText);
  });
}

function buttonClick(value) {
  isNaN(parseInt(value)) ? handleSymbol(value) : handleNumber(value);
}

init();
```