MANUAL TECNICO – CONJANALIZER

NOMBRE: KENETH WILLARD LOPEZ OVALLE

CARNÉ: 202100106

DESCRIPCION

Este manual técnico proporciona una guía detallada para el desarrollo, mantenimiento y actualización del proyecto de análisis de conjuntos y operaciones. Incluye información sobre la arquitectura del proyecto, el uso de herramientas y tecnologías, y una descripción de las clases y métodos principales.

INFORMACION GENERAL DEL PROYECTO

Descripción del Proyecto

El Proyecto de Análisis de Conjuntos y Operaciones es una aplicación desarrollada en Java que permite la definición y manipulación de conjuntos mediante operaciones algebraicas. El sistema ofrece una interfaz gráfica para la interacción del usuario, permitiendo el análisis léxico y sintáctico de entradas definidas por el usuario, y la visualización gráfica de las operaciones a través de diagramas de Venn. Además, genera reportes detallados de tokens y errores léxicos y sintácticos encontrados durante el análisis del código.

La aplicación puede ser utilizada para realizar operaciones clásicas de teoría de conjuntos, como unión, intersección, diferencia y complemento. Estas operaciones pueden ser visualizadas gráficamente, lo que permite al usuario tener una representación visual de los resultados.

Objetivos del Proyecto

1. **Análisis de conjuntos:** Permitir al usuario definir conjuntos y realizar operaciones entre ellos.
2. **Visualización gráfica**: Ofrecer una representación gráfica mediante diagramas de Venn que visualicen los resultados de las operaciones.
3. **Análisis Léxico y Sintáctico**: Proveer análisis léxicos y sintácticos de las entradas proporcionadas por el usuario, con generación automática de reportes de tokens y errores.
4. **Facilidad de uso:** Implementar una interfaz gráfica de usuario (GUI) intuitiva y fácil de manejar.
5. **Flexibilidad:** Permitir al usuario ejecutar múltiples operaciones de manera eficiente y navegar entre ellas.

Requisitos del Sistema

- Sistema Operativo: Windows, macOS, Linux
- Java Runtime Environment (JRE): Versión 8 o superior
- Espacio en Disco: Al menos 50 MB libres
- Memoria RAM: Mínimo 512 MB
- Dependencias:
    - JFlex: Herramienta de generación de analizadores léxicos.
    - CUP: Herramienta de análisis sintáctico para la generación de parsers en Java.
    - Swing: Biblioteca gráfica de Java para la creación de interfaces gráficas.

Tecnologías Utilizadas

Lenguaje de Programación:

El proyecto está desarrollado íntegramente en Java. Se ha utilizado Java debido a su portabilidad, facilidad de gestión de interfaces gráficas mediante Swing, y la disponibilidad de herramientas como Jflex y CUP para el análisis léxico y sintáctico.

1. **Herramientas y Librerías:**
    - Jflex: Generador de analizadores léxicos que toma como entrada un archivo .jflex para crear un analizador que identifica tokens en el texto ingresado por el usuario.
    - CUP (Construction of Useful Parsers): Herramienta para la generación de parsers a partir de una gramática definida en un archivo .cup.
    - Java Swing: Utilizado para desarrollar la interfaz gráfica, proporcionando componentes como Jframe, Jpanel, JtextArea, y Jbutton para la interacción del usuario.
    - AWT (Abstract Window Toolkit): Complementa el uso de Swing para el manejo de eventos gráficos.
2. **Entorno de Desarrollo:**
    - NetBeans IDE: IDE utilizado para el desarrollo del proyecto, con herramientas integradas para el diseño de interfaces gráficas y la depuración del código.
    - Git: Para el control de versiones y la colaboración en el proyecto.

Funcionalidades Clave del Proyecto

- **Definición de Conjuntos:** El usuario puede ingresar conjuntos mediante la interfaz, que luego son procesados por el analizador léxico y sintáctico para validar su formato.
- **Operaciones sobre Conjuntos:** Se pueden realizar operaciones algebraicas clásicas como:
    - **Unión:** Combina todos los elementos de los conjuntos.
    - **Intersección:** Obtiene los elementos comunes entre los conjuntos.
    - **Diferencia:** Retorna los elementos que están en un conjunto pero no en el otro.
    - **Complemento:** Retorna los elementos que no están en un conjunto respecto a un universo.
- **Visualización de Resultados:** Los resultados de las operaciones se representan visualmente mediante diagramas de Venn en un panel gráfico.
- **Análisis Léxico y Sintáctico:** El código ingresado por el usuario es procesado mediante un analizador léxico y un parser generado con Jflex y CUP. Se genera un informe con los tokens identificados y los errores encontrados durante el análisis.
- **Generación de Reportes:** Los resultados de los análisis léxicos y sintácticos se exportan en formato HTML, permitiendo al usuario revisar los errores léxicos y sintácticos en detalle.

DESCRIPCION DE CLASES Y METODOS PRINCIPALES

Clase Inicio

Descripción

La clase Inicio es la ventana principal de la aplicación y controla la interfaz gráfica de usuario (GUI). Utiliza la biblioteca Swing para gestionar la entrada del usuario, la visualización de resultados y los eventos como guardar archivos y ejecutar el análisis. Además, coordina las interacciones entre los diferentes componentes del sistema, como el analizador léxico (Lexer), el parser (Parser), y el gestor de conjuntos (ConjuntoManager).

Métodos Importantes

- **initComponents():**
  - Inicializa todos los componentes de la interfaz gráfica, como el área de texto para la entrada, la consola de salida, los botones, y el panel de diagramas de Venn.
- **AnalisisExcecuteActionPerformed(java.awt.event.ActionEvent evt):**
  - Ejecuta el análisis léxico y sintáctico del texto ingresado por el usuario, utilizando el Lexer y el Parser.
  - Genera reportes de tokens y errores, y actualiza la visualización del diagrama de Venn.
- **saveToFile(File file):**
  - Guarda el contenido del área de entrada en un archivo de texto en formato .ca.
- **updateVennDiagramPanel():**
  - Actualiza el panel de diagramas de Venn con los resultados de las operaciones de conjuntos realizadas.
- **TokensActionPerformed(java.awt.event.ActionEvent evt):**
  - Abre el reporte de tokens generados después del análisis léxico.

Clase Lexer

Descripción

La clase Lexer es generada automáticamente a partir del archivo Lexer.jflex utilizando JFlex. Se encarga de realizar el análisis léxico del texto ingresado por el usuario, dividiendo el código en tokens (unidades léxicas) que luego serán procesadas por el analizador sintáctico (parser).

Métodos Importantes

- **next_token():**
  - Retorna el siguiente token del flujo de entrada, identificando palabras clave, identificadores, operadores y otros elementos del lenguaje.
- **getTokens():**
  - Devuelve una lista de todos los tokens encontrados durante el análisis léxico.
- **getErrors():**
  - Retorna una lista de errores léxicos encontrados durante el análisis del código de entrada.

Clase Parser

Descripción

La clase Parser es generada automáticamente a partir del archivo Parser.cup utilizando **CUP**. Se encarga del análisis sintáctico, validando la estructura del código según las reglas de la gramática y generando un árbol de sintaxis. Trabaja en conjunto con el Lexer para procesar los tokens generados.

Métodos Importantes

- **parse()**:
    - Ejecuta el análisis sintáctico, procesando los tokens generados por el Lexer y validando la estructura del código.
- **syntax_error(Symbol s)**:
    - Manejador de errores sintácticos. Registra los errores y muestra mensajes de error en la consola y en los reportes.
- **getSyntaxErrors()**:
    - Retorna una lista de los errores sintácticos encontrados durante el análisis.

Clase ConjuntoManager

Descripción

La clase ConjuntoManager es responsable de la administración y operación de los conjuntos definidos en la aplicación. Almacena los conjuntos definidos por el usuario y proporciona funcionalidades para realizar operaciones algebraicas sobre ellos, como unión, intersección, diferencia, y complemento. Además, esta clase gestiona los resultados de las operaciones y los nodos correspondientes de los árboles de expresiones.

Métodos Importantes

- **definirConjunto(String nombre, Set<Character> conjunto):**
- Define un nuevo conjunto con el nombre proporcionado y lo almacena en la colección de conjuntos.
- **obtenerConjunto(String nombre):**
- Devuelve el conjunto asociado al nombre dado si este está definido.
- **guardarOperacion(String nombre, String notacion, Set<Character> conjuntoResultante):**
- Guarda una operación ejecutada con su notación y el conjunto resultante. También gestiona la relación con los nodos de operación (árbol de expresiones).
- **obtenerResultadoOperacion(String nombre):**
- Devuelve el resultado de una operación previamente realizada.

Clase Interna Operacion:

Esta clase interna representa una operación realizada sobre los conjuntos. Almacena la notación de la operación, el conjunto resultante, y el nodo de la operación en el árbol de expresiones.

Métodos Clave:

- getNotacion(): Devuelve la notación de la operación realizada.
- getConjuntoResultante(): Retorna el conjunto que resulta de la operación.
- getNodo(): Devuelve el nodo de la operación en el árbol de expresiones.

Clase GestorOperaciones

Descripción

La clase GestorOperaciones contiene las funciones estáticas necesarias para realizar las operaciones algebraicas sobre conjuntos, como unión, intersección, diferencia, y complemento. Estas funciones operan directamente sobre los objetos de tipo Set<Character>, devolviendo los resultados de cada operación.

Métodos Importantes

- **union(Set<Character> conjuntoA, Set<Character> conjuntoB):**
  - Realiza la operación de unión entre dos conjuntos. Devuelve un conjunto que contiene todos los elementos presentes en conjuntoA o conjuntoB.
- **interseccion(Set<Character> conjuntoA, Set<Character> conjuntoB):**
  - Realiza la operación de intersección entre dos conjuntos. Devuelve un conjunto con los elementos comunes entre conjuntoA y conjuntoB.
- **diferencia(Set<Character> conjuntoA, Set<Character> conjuntoB):**
  - Realiza la operación de diferencia entre dos conjuntos. Devuelve un conjunto con los elementos de conjuntoA que no están presentes en conjuntoB.
- **complemento(Set<Character> conjunto, Set<Character> universo)**
  - Calcula el complemento de un conjunto respecto a un conjunto universal. Devuelve un conjunto que contiene los elementos del universo que no están en el conjunto original.

Clase ArbolExpresion

Descripción

La clase ArbolExpresion es la estructura principal para representar un árbol de expresiones que maneja operaciones sobre conjuntos. Cada nodo del árbol representa una operación o un conjunto. La clase permite construir, simplificar, evaluar y visualizar el contenido del árbol.

Métodos Importantes

- **ArbolExpresion(Nodo raiz):**
  - Constructor que inicializa el árbol con una raíz dada.
- **getRaiz():**
  - Devuelve el nodo raíz del árbol.
- **setRaiz(Nodo raiz):**
  - Asigna el nodo raíz del árbol.
- **simplificar(SimplificadorOperaciones simplificador, String nombreOperacion):**
  - Simplifica el árbol de expresión utilizando el objeto SimplificadorOperaciones.
- **evaluar():**
  - Evalúa el árbol y devuelve el conjunto resultante de la operación que representa el árbol.
- **mostrarContenido():**
  - Retorna una representación en cadena de la expresión contenida en el árbol.

Clase Nodo

Descripción

Nodo es una clase abstracta que representa un nodo en el árbol de expresión. Un nodo puede ser una operación (unaria o binaria) o un conjunto. Esta clase define los métodos fundamentales que deben implementar las subclases.

Métodos Importantes

- **evaluar():**
  - Método abstracto que, al implementarse, debe evaluar el nodo y devolver el conjunto correspondiente.
- **mostrarContenido():**
  - Método abstracto que devuelve una representación textual del contenido del nodo.
- **calcularProfundidad():**
  - Método abstracto que calcula la profundidad del nodo en el árbol.
- **contarNodos():**
  - Método abstracto que cuenta el número de nodos en el árbol.
- **calcularComplejidad():**
  - Calcula la complejidad del nodo como la suma de la profundidad y el número de nodos.

Clase NodoBinario

Descripción

NodoBinario es una subclase de Nodo que representa una operación binaria (como la unión, intersección o diferencia) entre dos conjuntos o subexpresiones.

Métodos Importantes

- **NodoBinario(String operador, Nodo operandoIzquierdo, Nodo operandoDerecho):**
  - Constructor que inicializa la operación binaria con un operador y dos operandos.
- **evaluar():**
  - Evalúa la operación binaria entre los dos operandos, utilizando el operador especificado (U para unión, & para intersección, - para diferencia).
- **mostrarContenido():**
  - Devuelve una representación en cadena de la operación binaria.

Clase NodoUnario

Descripción

NodoUnario es una subclase de Nodo que representa una operación unaria (como el complemento) sobre un único conjunto o subexpresión.

Métodos Importantes

- **NodoUnario(String operador, Nodo operand):**
  - Constructor que inicializa la operación unaria con un operador y un operando.
- **evaluar():**
  - Evalúa la operación unaria sobre el operando, utilizando el operador especificado (^ para complemento).
- **mostrarContenido():**
  - Devuelve una representación en cadena de la operación unaria.

Clase NodoConjunto

Descripción

NodoConjunto es una subclase de Nodo que representa un conjunto. Es un nodo hoja en el árbol de expresiones, ya que no realiza ninguna operación, sino que simplemente almacena un conjunto de caracteres.

Métodos Importantes

- **NodoConjunto(String id, ConjuntoManager conjuntoManager):**
  - Constructor que inicializa el nodo con el identificador de un conjunto y un gestor de conjuntos (ConjuntoManager).
- **evaluar():**
  - Devuelve el conjunto almacenado en este nodo.
- **mostrarContenido():**
  - Retorna el identificador del conjunto como su representación textual.

Clase SimplificadorOperaciones

Descripción

SimplificadorOperaciones se encarga de aplicar leyes algebraicas (como la ley de absorción, conmutativa o de De Morgan) para simplificar los árboles de expresión. Utiliza una lista de leyes predefinidas para transformar el árbol en una versión más simple, sin cambiar el resultado de la operación.

Métodos Importantes

- **SimplificadorOperaciones(ConjuntoManager conjuntoManager):**
  - Constructor que inicializa el simplificador con el gestor de conjuntos.
- **simplificar(Nodo nodo, String nombreOperacion):**
  - Simplifica el nodo utilizando las leyes algebraicas aplicables y devuelve el nodo simplificado.
- **generarJSON(String nombreArchivo):**
  - Genera un archivo JSON con las operaciones y sus simplificaciones aplicadas.

Clase VennDiagramPanel

Descripción

La clase VennDiagramPanel es un componente gráfico de Swing que extiende JPanel y está diseñado para representar visualmente los conjuntos y las operaciones entre ellos en forma de diagramas de Venn. La clase toma como entrada un ArbolExpresion, que representa las operaciones entre conjuntos, y utiliza ese árbol para generar la visualización gráfica de las intersecciones, uniones y diferencias entre conjuntos.

Esta clase utiliza elementos de la biblioteca AWT y geom2d para el manejo gráfico de formas geométricas como elipses (que representan los conjuntos) y el área (que define las operaciones de intersección o unión).

Métodos Importantes

- **VennDiagramPanel(ArbolExpresion arbolExpresion, ConjuntoManager conjuntoManager):**
    - Constructor que inicializa el panel con el árbol de expresión (ArbolExpresion) y el gestor de conjuntos (ConjuntoManager). Configura las propiedades iniciales del panel, como el tamaño del "universo" en el que se dibujan los conjuntos.
- **paintComponent(Graphics g):**
    - Sobrescribe el método paintComponent de JPanel para realizar el dibujo de los conjuntos y sus operaciones en el panel. Utiliza Graphics2D para dibujar formas geométricas (elipses) y aplicar las operaciones correspondientes.
- **Pasos Clave:**
    1. Configura las propiedades de renderizado, como el suavizado de bordes.
    2. Dibuja el "universo" como un rectángulo que contiene los conjuntos.
    3. Calcula y dibuja las elipses que representan cada conjunto.
    4. Muestra las intersecciones o uniones entre conjuntos, basándose en el ArbolExpresion.
- **setArbolExpresion(ArbolExpresion arbolExpresion):**
    - Asigna un nuevo árbol de expresión al panel para ser visualizado. Llama a repaint() para actualizar la visualización con el nuevo árbol.

- **clear():**
    - Limpia el panel de cualquier diagrama de Venn existente, restableciendo el espacio de dibujo.
- **drawConjunto(Graphics2D g2d, Area areaConjunto, Color color, String etiqueta):**
    - Dibuja un conjunto específico en el panel.
- **Parámetros:**
    - g2d: Objeto Graphics2D utilizado para dibujar.
    - areaConjunto: Área geométrica del conjunto a dibujar.
    - color: Color del conjunto.
    - etiqueta: Etiqueta o nombre del conjunto que se dibujará junto a la forma.
- **createAreaForConjunto(String nombreConjunto):**
    - Crea un objeto Area (geométrica) para un conjunto específico, basada en su nombre. La Area se utilizará para operaciones posteriores como intersecciones o uniones.

Atributos Importantes

- **arbolExpresion:**
    - Referencia al árbol de expresión que representa las operaciones entre conjuntos.
- **conjuntoManager:**
    - Objeto que gestiona los conjuntos y permite obtener los conjuntos definidos para ser visualizados.
- **areasConjuntos:**
    - Mapa que asocia los nombres de los conjuntos con las áreas geométricas que los representan en el diagrama de Venn.
- **universeWidth, universeHeight, universeX, universeY:**
    - Definen las dimensiones y la posición del rectángulo que representa el "universo" en el que se encuentran los conjuntos.

ANEXOS

```java
public class Inicio extends javax.swing.JFrame {
    public List<ArbolExpresion> arbolesExpresion = new ArrayList<>();
    private OutputManager outputManager;
    private ConjuntoManager conjuntoManager;
    private SimplificadorOperaciones simplificador;
    private VennDiagramPanel vennDiagramPanel;
    private ArbolExpresion arbolExpresion;
    private List<Map.Entry<String, Operacion>> operacionesList;
    private int currentOperationIndex = 0;
    private Parser parser;  // Añadir aquí


    /**
     * Creates new form Inicio
     */
    /**
     * Creates new form Inicio
     */
    public Inicio() {
        this.outputManager = new OutputManager();
        this.conjuntoManager = new ConjuntoManager();
        this.simplificador = new SimplificadorOperaciones(conjuntoManager);
        Nodo nodoRaizInicial = new NodoConjunto("", conjuntoManager);
        // Inicializar el árbol de expresiones con el nodo raíz inicial
        this.arbolExpresion = new ArbolExpresion(nodoRaizInicial);
        initComponents();
        // Inicializar el VennDiagramPanel con el árbol de expresión
        vennDiagramPanel = new VennDiagramPanel(arbolExpresion, conjuntoManager);
        JpanelGraph.setLayout(new BorderLayout());
        JpanelGraph.add(vennDiagramPanel, BorderLayout.CENTER);
        JpanelGraph.revalidate();
        JpanelGraph.repaint();
    }
```

Figura 1. *Definición de clase inicio.*

Fuente: *Elaboración propia 2024*

```java
public class ConjuntoManager {
    // Clase interna que representa una operación
    public static class Operacion {
        private String notacion;
        private Set<Character> conjuntoResultante;
        private Nodo nodoOperacion;

        public Operacion(String notacion, Set<Character> conjuntoResultante) {
            this.notacion = notacion;
            this.conjuntoResultante = conjuntoResultante;
        }

        public String getNotacion() {
            return notacion;
        }

        public Set<Character> getConjuntoResultante() {
            return conjuntoResultante;
        }

        public Nodo getNodo() {
            return nodoOperacion;
        }
    }

    private Map<String, Set<Character>> conjuntos;
    private Map<String, Operacion> operaciones;

    public ConjuntoManager() {
        this.conjuntos = new HashMap<>();
        this.operaciones = new HashMap<>();
    }
```

Figura 2. *Definición de clase ConjuntoManager.*

Fuente: *Elaboración propia 2024*

```java
public class GestorOperaciones {

    // Unión de dos conjuntos
    public static Set<Character> union(Set<Character> conjuntoA, Set<Character> conjuntoB) {
        Set<Character> resultado = new HashSet<>(conjuntoA);
        resultado.addAll(conjuntoB);
        return resultado;
    }

    // Intersección de dos conjuntos
    public static Set<Character> interseccion(Set<Character> conjuntoA, Set<Character> conjuntoB) {
        Set<Character> resultado = new HashSet<>(conjuntoA);
        resultado.retainAll(conjuntoB);
        return resultado;
    }

    // Diferencia de dos conjuntos (A - B)
    public static Set<Character> diferencia(Set<Character> conjuntoA, Set<Character> conjuntoB) {;
        Set<Character> resultado = new HashSet<>(conjuntoA);
        resultado.removeAll(conjuntoB);
        return resultado;
    }

    // Complemento de un conjunto respecto al conjunto universal
    public static Set<Character> complemento(Set<Character> conjunto) {
        Set<Character> conjuntoUniversal = generarConjuntoUniversal();
        conjuntoUniversal.removeAll(conjunto);
        return conjuntoUniversal;
    }

    // Genera el conjunto universal (caracteres ASCII del 33 al 126)
    private static Set<Character> generarConjuntoUniversal() {
        Set<Character> conjuntoUniversal = new HashSet<>();
        for (char c = '!'; c <= '~'; c++) {
            conjuntoUniversal.add(c);
        }
        return conjuntoUniversal;
    }
}
```

Figura 3. *Definición de clase GestorOperaciones.*

Fuente: *Elaboración propia 2024*

```java
public class ArbolExpresion {
    private Nodo raiz;
    private Nodo raizSimplificada;

    public ArbolExpresion(Nodo raiz) {
        this.raiz = raiz;
    }

    public ArbolExpresion() {
        this.raiz = null;
    }

    public Nodo getRaiz() {
        return raiz;
    }

    public void setRaiz(Nodo raiz) {
        this.raiz = raiz;
    }

    public void simplificar(SimplificadorOperaciones simplificador, String nombreOperacion) {
        this.raiz = simplificador.simplificar(this.raiz, nombreOperacion);
    }

    public Set<Character> evaluar() {
        return raiz.evaluar();
    }

    public String mostrarContenido() {
        return raiz.mostrarContenido();
    }

    public void construir(Nodo nodoRaiz) {
        this.raiz = nodoRaiz;
    }

    public void setRaizSimplificada(Nodo raizSimplificada) {
        this.raizSimplificada = raizSimplificada;
    }

    public Nodo getRaizSimplificada() {
        return raizSimplificada;
    }
```

Figura 4. *Definición de clase ArbolExpresion.*

Fuente: *Elaboración propia 2024*

```java
public abstract class Nodo {
    private boolean simplificado = false;

    // Método abstracto para evaluar el nodo y obtener su conjunto de caracteres
    public abstract Set<Character> evaluar();

    // Método abstracto para mostrar el contenido del nodo como una cadena
    public abstract String mostrarContenido();

    // Método abstracto para representar el nodo como una cadena
    @Override
    public abstract String toString();

    public boolean isSimplificado() {
        return simplificado;
    }

    public void setSimplificado(boolean simplificado) {
        this.simplificado = simplificado;
    }
    //método para calcular la profundidad del nodo
    public abstract int calcularProfundidad();
    public abstract int contarNodos();

    public int calcularComplejidad() {
        return calcularProfundidad() + contarNodos();
    }

    //método abstracto para recopilar nombres de conjuntos en el nodo y sus hijos
    public abstract void recopilarConjuntos(Set<String> conjuntos);
}
```

Figura 5. *Definición de clase abstracta Nodo.*

Fuente: *Elaboración propia 2024*

```java
public class NodoBinario extends Nodo {
    private String operador;
    private Nodo izquierdo;
    private Nodo derecho;
    private Area areaCache;

    public NodoBinario(String operador, Nodo izquierdo, Nodo derecho) {
        if (operador == null || izquierdo == null || derecho == null) {
            throw new IllegalArgumentException("Operador o nodos no pueden ser nulos");
        }
        this.operador = operador;
        this.izquierdo = izquierdo;
        this.derecho = derecho;
        this.areaCache = null;
    }

    @Override
    public Set<Character> evaluar() {
        Set<Character> resultadoIzquierdo = izquierdo.evaluar();
        Set<Character> resultadoDerecho = derecho.evaluar();

        switch (operador) {
            case "U":
                return GestorOperaciones.union(resultadoIzquierdo, resultadoDerecho);
            case "&":
                return GestorOperaciones.interseccion(resultadoIzquierdo, resultadoDerecho);
            case "-":
                return GestorOperaciones.diferencia(resultadoIzquierdo, resultadoDerecho);
            default:
                throw new IllegalArgumentException("Operador binario desconocido: " + operador);
        }
    }
    @Override
    public String mostrarContenido() {
        return "(" + izquierdo.mostrarContenido() + " " + operador + " " + derecho.mostrarContenido() + ")";
    }

    public Shape obtenerForma() {
        return areaCache;
    }

    @Override
    public String toString() {
        return mostrarContenido();
    }
```

Figura 6. *Definición de clase NodoBinario.*

Fuente: *Elaboración propia 2024*

```java
public class NodoConjunto extends Nodo {
    private String nombreConjunto;
    private ConjuntoManager conjuntoManager;
    private Area areaCache;

    public NodoConjunto(String nombreConjunto, ConjuntoManager conjuntoManager) {
        this.nombreConjunto = nombreConjunto;
        this.conjuntoManager = conjuntoManager;
        this.areaCache = null; // Inicialmente nula, para calcularla después si es necesario
    }

    @Override
    public Set<Character> evaluar() {
        return conjuntoManager.obtenerConjunto(nombreConjunto);
    }

    @Override
    public String mostrarContenido() {
        return nombreConjunto;
    }

    @Override
    public int contarNodos() {
        return 1; // NodoConjunto es una hoja
    }

    @Override
    public String toString() {
        return nombreConjunto;
    }

    public String getNombreConjunto() {
        return nombreConjunto;
    }

    @Override
    public void recopilarConjuntos(Set<String> conjuntos) {
        conjuntos.add(nombreConjunto);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        NodoConjunto that = (NodoConjunto) obj;
        return nombreConjunto.equals(that.nombreConjunto);
    }
```

Figura 7. *Definición de clase NodoConjunto.*

Fuente: *Elaboración propia 2024*

```java
public class NodoUnario extends Nodo {
    private String operador;
    private Nodo operand;


    public NodoUnario(String operador, Nodo operand) {
        if (operador == null || operand == null) {
            throw new IllegalArgumentException("Operador o operando no pueden ser nulos");
        }
        this.operador = operador;
        this.operand = operand;
    }

    @Override
    public Set<Character> evaluar() {
        Set<Character> resultadoOperand = operand.evaluar();

        switch (operador) {
            case "^":
                return GestorOperaciones.complemento(resultadoOperand);
            default:
                throw new IllegalArgumentException("Operador unario desconocido: " + operador);
        }
    }

    @Override
    public String mostrarContenido() {
        return operador + operand.mostrarContenido();
    }

    @Override
    public void recopilarConjuntos(Set<String> conjuntos) {
        operand.recopilarConjuntos(conjuntos);
    }

    @Override
    public String toString() {
        return mostrarContenido();
    }

    public String getOperador() {
        return operador;
    }

    public void setOperador(String operador) {
        this.operador = operador;
    }
```

Figura 8. *Definición de clase NodoUnario.*

Fuente: *Elaboración propia 2024*

```java
public class SimplificadorOperaciones {
    private ConjuntoManager conjuntoManager;
    private Map<String, List<String>> leyesAplicadasPorOperacion;
    private List<Ley> leyes;
    private static final int MAX_ITERACIONES = 50;

    public SimplificadorOperaciones(ConjuntoManager conjuntoManager) {
        this.conjuntoManager = conjuntoManager;
        this.leyesAplicadasPorOperacion = new HashMap<>();
        this.leyes = Arrays.asList(
            new Absorcion(),
            new Asociativa(),
            new Conmutativa(),
            new DeMorgan(),
            new Distributiva(),
            new DobleComplemento(),
            new Idempotente()
        );
    }
```

```java
public Nodo simplificar(Nodo nodo, String nombreOperacion) {
    Set<String> estadosVisitados = new HashSet<>();
    Map<String, Nodo> cacheSimplificaciones = new HashMap<>();
    List<String> leyesAplicadas = new ArrayList<>();
    boolean cambios;
    int iteraciones = 0;

    int complejidadInicial = nodo.calcularComplejidad();
    int maxIteracionesAdaptativo = Math.max(complejidadInicial * 10, MAX_ITERACIONES);
    int maxSinProgreso = 5;
    int sinProgreso = 0;
    int complejidadPrev = complejidadInicial;
    Nodo mejorSimplificacion = nodo;

    do {
        cambios = false;
        Nodo resultadoSimplificado = simplificarRecursivo(nodo, estadosVisitados, cacheSimplificaciones, leyesAplicadas);

        if (!resultadoSimplificado.equals(nodo)) {
            cambios = true;
            nodo = resultadoSimplificado;

            int nuevaComplejidad = nodo.calcularComplejidad();

            // Control de expansión
            if (nuevaComplejidad > complejidadPrev) {
                sinProgreso++;
                if (sinProgreso >= maxSinProgreso) {
                    return mejorSimplificacion; // Revertir a la mejor simplificación
                }
            } else {
                sinProgreso = 0; // Reinicia si hay progreso
                mejorSimplificacion = nodo; // Actualiza la mejor simplificación
            }

            complejidadPrev = nuevaComplejidad; // Actualiza la complejidad anterior
        }

        iteraciones++;
    } while (cambios && iteraciones < maxIteracionesAdaptativo);

    leyesAplicadasPorOperacion.put(nombreOperacion, leyesAplicadas);
    return nodo;
}
```

Figura 8. *Definición de clase simplificadorOperaciones.*

Fuente: *Elaboración propia 2024*