ESCUELA POLITÉCNICA
SUPERIOR DE CÓRDOBA
Universidad de Córdoba

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

SimAS 3.0 descendente predictivo. Simulador de analizadores sintácticos descendentes predictivos.

Manual de Código

Autor
D. Antonio Llamas García

Directores
Prof. Dr. Antonio Araúzo Azofra
Prof. Dra. María Luque Rodríguez

Septiembre, 2025

UNIVERSIDAD D CÓRDOBA

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Propósito del documento

Este manual de código tiene como objetivo proporcionar una documentación técnica completa y detallada de la aplicación **SimAS 3.0** (Simulador de Análisis Sintáctico), desarrollada como Trabajo de Fin de Grado en Ingeniería Informática en la Universidad de Córdoba.

El documento está dirigido a desarrolladores, mantenedores del software y cualquier persona que necesite comprender la estructura interna, el funcionamiento y los algoritmos implementados en la aplicación. A diferencia de versiones anteriores del manual, este documento no incluye código fuente embebido, sino que hace referencia al repositorio oficial donde se encuentra todo el código fuente completo y actualizado.

1.2. Descripción general del proyecto

SimAS 3.0 es una aplicación educativa desarrollada en Java que implementa un simulador de analizadores sintácticos descendentes predictivos. La aplicación permite a los usuarios:

- Crear y editar gramáticas libres de contexto
- Generar automáticamente tablas de análisis predictivo
- Simular el proceso de análisis sintáctico descendente
- Visualizar el árbol de derivación resultante
- Gestionar funciones de error personalizadas

1.3. Repositorio oficial del proyecto

Todo el código fuente de SimAS 3.0 está disponible en el repositorio oficial de GitHub:

https://github.com/Llamatekee/SimAS-3.0.git

1.3.1. Estructura del repositorio

El repositorio está organizado de la siguiente manera:

- `src/`: Directorio principal del código fuente
- `lib/`: Librerías externas (JavaFX SDK)
- `build/`: Scripts de construcción
- `dist-standalone/`: Ejecutables generados
- `resources/`: Recursos gráficos e internacionalización
- `vistas/`: Archivos FXML de la interfaz

1.3.2. Acceso al código fuente

Para acceder al código fuente completo:

1. Clonar el repositorio:

```
1     git clone https://github.com/Llamatekee/SimAS-3.0.git
2
```

2. Explorar la estructura de paquetes en `src/`
3. Revisar la documentación de clases en los archivos Java
4. Consultar los archivos de configuración y scripts de construcción

1.4. Características principales

1.4.1. Interfaz de usuario moderna

La aplicación utiliza JavaFX 17 para proporcionar una interfaz de usuario moderna e intuitiva, con:

- Diseño responsivo y adaptable

- Sistema de pestañas para múltiples proyectos
- Interfaz completamente internacionalizada (6 idiomas)
- Atajos de teclado para operaciones frecuentes

1.4.2. Algoritmos de análisis sintáctico

Implementa algoritmos estándar de análisis sintáctico descendente predictivo:

- Construcción de conjuntos FIRST y FOLLOW
- Generación de tablas de análisis predictivo
- Algoritmo de análisis LL(1)
- Detección y manejo de conflictos
- Funciones de error personalizables

1.4.3. Arquitectura modular

El sistema está diseñado con una arquitectura modular que separa claramente:

- Lógica de negocio (modelo de gramáticas)
- Interfaz de usuario (controladores y vistas)
- Algoritmos de análisis (simulador)
- Utilidades y servicios auxiliares

1.5. Alcance del manual

Este manual cubre los siguientes aspectos de la aplicación:

1. **Acceso al repositorio**: cómo obtener y navegar el código fuente.
2. **Arquitectura del sistema**: diseño general y organización de componentes.
3. **Documentación de paquetes**: descripción detallada de cada paquete Java.
4. **Clases principales**: documentación de las clases más importantes.
5. **Algoritmos implementados**: explicación de los algoritmos de análisis sintáctico.
6. **Interfaz de usuario**: documentación de la capa de presentación.
7. **Sistema de internacionalización**: gestión de múltiples idiomas.
8. **Compilación y despliegue**: proceso de construcción de la aplicación.
9. **Convenciones y mejores prácticas**: estándares de desarrollo utilizados.

1.6. Convenciones utilizadas

A lo largo de este manual se utilizarán las siguientes convenciones:

- **Nombres de clases**: se escribirán en formato `ClaseNombre`.
- **Nombres de métodos**: se escribirán en formato `metodoNombre()`.
- **Paquetes**: se escribirán en formato `paquete.subpaquete`.
- **Archivos**: se escribirán en formato `archivo.extensión`.
- **Rutas del repositorio**: referencias a archivos usando rutas relativas al repositorio.
- **Enlaces a GitHub**: enlaces directos a archivos específicos en el repositorio.

1.7. Estructura del documento

El manual está organizado en las siguientes secciones principales:

Parte I - Acceso y Arquitectura Contiene información sobre el repositorio, estructura general y arquitectura del sistema.

Parte II - Componentes del Sistema Incluye la documentación detallada de paquetes, clases y algoritmos.

Parte III - Desarrollo y Despliegue Cubre aspectos técnicos de desarrollo, internacionalización y despliegue.

Esta estructura permite tanto una visión general del sistema como referencias específicas al código fuente en el repositorio, facilitando la comprensión y mantenimiento del código.

1.8. Referencias técnicas

Este manual se basa en los principios fundamentales de análisis sintáctico descritos en [1] y [3], implementados utilizando las tecnologías Java [6] y JavaFX [5]. La aplicación utiliza herramientas modernas de desarrollo como [7] para la generación de ejecutables nativos.

Para información detallada sobre la implementación específica de algoritmos y estructuras de datos, consulte directamente el código fuente en el repositorio oficial.

Referencias al Manual Técnico: Para explicaciones más detalladas sobre el diseño arquitectónico, análisis de clases, métricas de complejidad y patrones de diseño implementados en SimAS 3.0, consulte:

- Capítulo 8: "Diseño de paquetes Arquitectura completa del sistema
- Capítulo 9: "Diseño de clases Implementación detallada de todas las clases

Parte I

Documentación externa

Capítulo 2

Recursos y Requisitos de desarrollo y ejecución

2.1. Introducción

Este capítulo describe los recursos necesarios para el desarrollo, compilación y ejecución de la aplicación SimAS 3.0. Se incluyen tanto los requisitos técnicos como las herramientas de desarrollo utilizadas en el proyecto.

2.2. Requisitos del sistema

2.2.1. Requisitos mínimos de hardware

Para la ejecución de SimAS 3.0 se requieren los siguientes recursos mínimos de hardware:

- **Procesador**: Intel Core i3 o equivalente AMD.
- **Memoria RAM**: 4 GB (recomendado 8 GB).
- **Espacio en disco**: 500 MB para la aplicación y dependencias.
- **Resolución de pantalla**: 1024x768 píxeles (recomendado 1366x768 o superior).
- **Conectividad**: no requerida para funcionamiento básico.

2.2.2. Requisitos de software

2.2.2.1. Java Runtime Environment (JRE)

- **Versión requerida**: Java 17 o superior.
- **Distribución**: Oracle JDK, OpenJDK, o cualquier distribución compatible.

- **Arquitectura**: Compatible con x64, ARM64 (Apple Silicon).

2.2.2.2. Sistemas operativos soportados

- **Windows**: Windows 10 o superior (x64).
- **macOS**: macOS 10.15 (Catalina) o superior.
- **Linux**: distribuciones modernas con soporte para JavaFX.

2.3. Entorno de desarrollo

2.3.1. Java Development Kit (JDK)

- **Versión**: JDK 17 o superior.
- **Recomendado**: OpenJDK 17 LTS.
- **Configuración**: variables de entorno `JAVA_HOME` configuradas correctamente.

2.3.2. JavaFX SDK

- **Versión**: JavaFX 17.0.12.
- **Ubicación**: Incluido en el directorio `lib/javafx-sdk-17.0.12/`.
- **Propósito**: proporciona las librerías necesarias para la interfaz gráfica.

2.3.3. Herramientas de construcción

- **jpackage**: incluido en JDK 14+ para crear ejecutables nativos.
- **Scripts de construcción**:
  - `build.sh` - Para sistemas Unix/Linux/macOS.
  - `build.bat` - Para sistemas Windows.
  - `create-standalone-app.sh` - Para crear aplicación independiente.

2.4. Arquitectura del sistema

2.4.1. Visión general

SimAS 3.0 sigue una arquitectura de capas bien definida que separa las responsabilidades de cada componente:

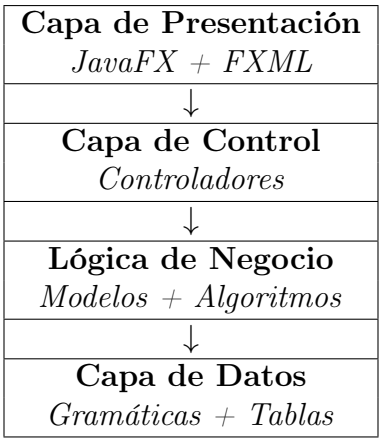

Figura 2.1: Arquitectura en capas de SimAS 3.0

2.4.2. Patrones de diseño utilizados

2.4.2.1. Modelo-Vista-Controlador (MVC)

- **Modelo**: clases en el paquete `gramatica` que representan las entidades del dominio.
- **Vista**: archivos FXML en el directorio `vistas`.
- **Controlador**: clases Java que manejan la lógica de presentación.

2.4.2.2. Observer Pattern

- Utilizado para la gestión de eventos de la interfaz de usuario.
- Implementado a través de los mecanismos de JavaFX.

2.4.2.3. Factory Pattern

- Para la creación de diferentes tipos de símbolos (terminales, no terminales).
- En la generación de componentes de la interfaz de usuario.

2.5. Organización modular

2.5.1. Estructura de paquetes

La aplicación está organizada en los siguientes paquetes principales:

`bienvenida` Gestión de la pantalla de bienvenida y menú principal.

`editor` Editor de gramáticas con múltiples paneles de configuración.

`gramatica` Modelo de datos y algoritmos para gramáticas libres de contexto.

`simulador` Simulador de análisis sintáctico descendente predictivo.

`utils` Utilidades generales, internacionalización y gestión de ventanas.

`centroayuda` Sistema de ayuda y documentación integrada.

`vistas` Archivos FXML que definen las interfaces de usuario.

2.5.2. Dependencias entre módulos

Módulo	**Dependencias**
bienvenida	editor, simulador, utils, gramatica
editor	gramatica, utils, vistas
simulador	gramatica, utils, vistas
gramatica	(independiente)
utils	(independiente)
centroayuda	utils
vistas	utils

Figura 2.2: Dependencias entre módulos de SimAS 3.0

2.6. Recursos de compilación

2.6.1. Scripts de construcción

2.6.1.1. build.sh (Unix/Linux/macOS)

- Compila el código fuente Java.
- Empaqueta las dependencias JavaFX.
- Genera el archivo JAR ejecutable.
- Crea la estructura de directorios necesaria.

2.6.1.2. build.bat (Windows)

- Equivalente al script build.sh para sistemas Windows.
- Utiliza comandos nativos de Windows.
- Genera la misma estructura de archivos.

2.6.1.3. create-standalone-app.sh

- Crea una aplicación independiente para macOS.
- Utiliza jpackage para generar un archivo .app.
- Incluye todas las dependencias necesarias.
- Genera un ejecutable completamente autónomo.

2.6.2. Configuración de compilación

El proceso de compilación requiere:

- Configuración correcta de las variables de entorno Java.
- Acceso a las librerías JavaFX en `lib/javafx-sdk-17.0.12/`.
- Permisos de ejecución en los scripts de construcción.
- Espacio suficiente en disco para archivos temporales.

Parte II

Documentación interna

Capítulo 3

Código de la aplicación

3.1. Introducción

Este capítulo presenta una documentación detallada del código fuente de SimAS 3.0, con referencias específicas al repositorio de GitHub donde se encuentra todo el código fuente completo. A diferencia de versiones anteriores, este capítulo no incluye fragmentos de código embebidos, sino referencias directas a los archivos en el repositorio.

Referencias al Manual Técnico: Para explicaciones más detalladas sobre las clases, paquetes y dependencias del sistema, incluyendo diagramas de clases completos, análisis de algoritmos, patrones de diseño implementados y métricas de complejidad, consulte:

- Capítulo 8: "Diseño de paquetes Arquitectura completa y análisis de dependencias.
- Capítulo 9: "Diseño de clases Implementación detallada de todas las clases principales.

3.2. Punto de entrada de la aplicación

3.2.1. Clase Bienvenida

La clase `Bienvenida` es el punto de entrada de la aplicación y gestiona la pantalla de bienvenida.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/bienvenida/Bienvenida.java

Funcionalidades principales:

- Punto de entrada principal de la aplicación JavaFX.
- Configuración de la ventana de bienvenida (sin decoraciones, dimensiones fijas).

- Timeline automática que transita al menú principal después de 2.5 segundos.
- Gestión del ciclo de vida inicial de la aplicación.

Métodos principales:

- `start(Stage)`: inicializa la interfaz de bienvenida.
- `abrirMenuPrincipal()`: transita al menú principal.
- `main(String[])`: método main estático.

3.2.2. Clase MenuPrincipal

La clase `MenuPrincipal` es el controlador principal de navegación de la aplicación.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/bienvenida/MenuPrincipal.java

Funcionalidades principales:

- Gestión completa de la interfaz principal de navegación.
- Sistema avanzado de pestañas para múltiples proyectos simultáneos.
- Soporte completo de internacionalización con cambio dinámico de idioma.
- Gestión de atajos de teclado para operaciones frecuentes.
- Coordinación entre módulos (editor, simulador, ayuda).
- Sistema de ventanas secundarias para gestión avanzada de pestañas.

Métodos principales:

- `start(Stage)`: inicializa la ventana principal.
- `cambiarIdioma()`: gestiona el cambio dinámico de idioma.
- `onBtnEditorAction()`: abre el editor de gramáticas.
- `onBtnSimuladorAction()`: abre el simulador.
- `onBtnAyudaAction()`: abre el sistema de ayuda.

3.3. Modelo de datos: Paquete gramatica

Referencia al Manual Técnico: Para una explicación más detallada del paquete gramatica, incluyendo diagramas de clases completos, análisis de algoritmos de complejidad $O(n^3)$, patrones de diseño implementados (Composite, Factory, Strategy) y todas las relaciones de dependencia, consulte:

- Capítulo 8: "Diseño de paquetes Arquitectura y dependencias del paquete gramatica.
- Capítulo 9: "Diseño de clases Implementación detallada de clases como Gramatica, Simbolo, Terminal, NoTerminal,

3.3.1. Clase Gramatica

La clase `Gramatica` es el núcleo del sistema de modelado de gramáticas, implementando los algoritmos fundamentales de análisis sintáctico.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Gramatica.java

Funcionalidades principales:

- Gestión completa de símbolos terminales y no terminales.
- Administración de producciones gramaticales con operaciones CRUD.
- Implementación de algoritmos de cálculo de conjuntos PRIMERO y SIGUIENTE.
- Generación automática de tablas de análisis predictivo.
- Validación de gramáticas LL(1) con detección de conflictos.
- Transformaciones gramaticales (eliminación de recursividad, factorización).
- Persistencia y carga de gramáticas desde archivos.
- Integración completa con JavaFX mediante propiedades observables.

Algoritmos implementados:

- Cálculo iterativo de conjuntos PRIMERO.
- Cálculo iterativo de conjuntos SIGUIENTE.
- Construcción de tablas predictivas $O(n^2)$.
- Eliminación de recursividad izquierda.
- Factorización automática.

3.3.2. Jerarquía de símbolos

3.3.2.1. Clase Simbolo (abstracta)

Clase base abstracta que define la interfaz común para todos los símbolos gramaticales.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Simbolo.java

Responsabilidades:

- Definición de interfaz común para símbolos.
- Gestión de nombres y valores de símbolos.
- Implementación de métodos de comparación e igualdad.
- Soporte para propiedades JavaFX observables.

3.3.2.2. Clase Terminal

Implementación concreta para símbolos terminales de la gramática.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Terminal.java

Características específicas:

- Representación de tokens del lenguaje.
- Métodos específicos para análisis léxico.
- Comparación y validación de terminales.
- Integración con el analizador sintáctico.

3.3.2.3. Clase NoTerminal

Implementación concreta para símbolos no terminales de la gramática.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/NoTerminal.java

Características específicas:

- Gestión de conjuntos PRIMERO y SIGUIENTE.
- Algoritmos de cálculo iterativo.
- Métodos de comparación específicos.
- Integración con algoritmos de análisis predictivo.

3.4. Algoritmos de análisis sintáctico

3.4.1. Cálculo de conjuntos PRIMERO

El algoritmo para calcular los conjuntos PRIMERO se implementa en la clase `Gramatica`. Este algoritmo calcula, para cada símbolo no terminal, el conjunto de símbolos terminales que pueden aparecer al inicio de las cadenas derivadas de ese símbolo.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Gramatica.java (método `calcularFirst()`)

Algoritmo implementado:

1. Inicializar conjuntos PRIMERO vacíos para todos los no terminales.
2. Mientras haya cambios en los conjuntos:
   a) Para cada producción $A \to \alpha$
   b) Calcular PRIMERO(α)
   c) Agregar PRIMERO(α) a PRIMERO(A)
3. Repetir hasta convergencia.

Complejidad: O(n^3) en el peor caso, donde n es el número de símbolos.

3.4.2. Cálculo de conjuntos SIGUIENTE

El algoritmo para calcular los conjuntos SIGUIENTE determina, para cada símbolo no terminal, el conjunto de símbolos terminales que pueden aparecer inmediatamente después de ese símbolo en alguna derivación.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Gramatica.java (método `calcularFollow()`)

Algoritmo implementado:

1. Inicializar conjuntos SIGUIENTE vacíos.
2. SIGUIENTE(S) = {\$} donde S es el símbolo inicial.
3. Mientras haya cambios en los conjuntos:
   a) Para cada producción $A \to \alpha B \beta$
   b) Agregar PRIMERO(β) a SIGUIENTE(B)
   c) Si β puede derivar ϵ, agregar SIGUIENTE(A) a SIGUIENTE(B)
4. Repetir hasta convergencia.

Complejidad: O(n^3) en el peor caso.

3.5. Clases del paquete simulador

Referencia al Manual Técnico: Para una explicación más detallada del paquete simulador, incluyendo algoritmos de análisis sintáctico descendente predictivo, manejo de errores con funciones personalizables, patrones de diseño (Strategy, Observer, Template Method) y análisis de complejidad algorítmica, consulte:

- Capítulo 8: "Diseño de paquetes Arquitectura completa del simulador.
- Capítulo 9: "Diseño de clases Implementación detallada de SimulacionFinal y clases relacionadas.

3.5.1. Clase PanelSimulacion

Panel básico que gestiona la interfaz simplificada del simulador.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/simulador/PanelSimulacion.java

Funcionalidades principales:

- Interfaz simplificada para demostraciones de análisis sintáctico.
- Visualización básica de pila, entrada y resultados.
- Controles básicos de navegación (siguiente paso).
- Integración con el sistema de pestañas.

3.5.2. Clase SimulacionFinal

La clase `SimulacionFinal` es el motor principal del simulador de análisis sintáctico descendente predictivo.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/simulador/SimulacionFinal.java

Características principales:

- **Algoritmo completo LL(1):** implementa el análisis sintáctico descendente predictivo con pila.
- **Simulación interactiva:** navegación paso a paso con avance, retroceso e ir al inicio/final.
- **Visualización en tiempo real:** estado de pila, entrada restante y acciones realizadas.
- **Generación de derivaciones:** construcción automática de la derivación izquierda.

- **Visualización de árboles:** generación de representaciones gráficas con Graphviz.
- **Manejo de errores:** sistema completo de recuperación de errores con funciones personalizables.
- **Historial completo:** registro detallado de todos los pasos de simulación.
- **Internacionalización:** soporte completo para 6 idiomas.
- **Informes PDF:** generación automática de informes profesionales.

Algoritmo principal:

El algoritmo implementado sigue estos pasos para cada avance en la simulación:

1. Verificar condiciones de aceptación (pila y entrada contienen $).
2. Si el símbolo en cima de pila coincide con el símbolo de entrada actual:
   a) Consumir símbolo de la entrada.
   b) Desapilar símbolo.
   c) Registrar acción de .emparejar".
3. En caso contrario:
   a) Consultar tabla predictiva.
   b) Aplicar producción o función de error correspondiente.
   c) Actualizar pila y registrar acción.
4. Actualizar interfaz y historial.

3.6. Clases del paquete editor

3.6.1. Clase Editor

La clase `Editor` es el componente principal para la creación y edición de gramáticas en SimAS 3.0.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/editor/Editor.java

Características principales:

- **Interfaz completa de edición:** proporciona una interfaz completa para crear y editar gramáticas.
- **Validación integrada:** incluye validación automática de gramáticas con mensajes de error detallados.

- **Generación de informes:** crea informes PDF profesionales de las gramáticas.
- **Sistema de pestañas jerárquico:** gestiona relaciones padre-hijo entre pestañas.
- **Internacionalización:** soporte completo para múltiples idiomas.
- **Integración con simulador:** permite lanzar simulaciones directamente desde el editor.
- **Guardado y carga:** soporta guardar y cargar gramáticas desde archivos.
- **Estados dinámicos:** actualiza automáticamente el estado de los botones según la gramática actual.

Constructor principal:

Inicializa el editor con configuración completa del sistema de pestañas y relaciones jerárquicas.

3.7. Clases del paquete utils

3.7.1. Clase TabManager

La clase `TabManager` es el núcleo del sistema de gestión de pestañas de SimAS 3.0, implementando un sistema complejo de pestañas jerárquicas.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/TabManager.java

Características principales:

- **Gestión de pestañas jerárquica:** sistema padre-hijo para organizar pestañas relacionadas.
- **Agrupación de elementos:** agrupa editores y simuladores relacionados por número de grupo.
- **Movimiento entre ventanas:** permite arrastrar y soltar pestañas entre ventanas.
- **Renovación automática:** reasigna números de grupo cuando cambian las pestañas.
- **Internacionalización:** soporte para textos en múltiples idiomas.
- **Instancias múltiples:** permite múltiples instancias de editores y simuladores.
- **Menús contextuales:** gestiona menús contextuales personalizados para pestañas.
- **Persistencia de estado:** mantiene el estado de grupos y relaciones entre sesiones.

Método principal de creación de pestañas:

Implementa lógica compleja para determinar si usar caché o crear nueva instancia basada en las reglas jerárquicas del sistema.

3.7.2. Clase SecondaryWindow

La clase `SecondaryWindow` implementa el sistema de ventanas secundarias de SimAS 3.0.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/SecondaryWindow.java

Características principales:

- **Ventanas múltiples:** permite crear múltiples ventanas secundarias independientes.
- **Arrastre de pestañas:** soporta arrastrar y soltar pestañas entre ventanas.
- **Atajos de teclado:** implementa atajos específicos para ventanas secundarias.
- **Numeración automática:** asigna números automáticamente a las ventanas.
- **Internacionalización:** soporte completo para múltiples idiomas.
- **Integración con TabManager:** funciona perfectamente con el sistema de pestañas.
- **Cierre inteligente:** gestiona el cierre automático cuando no hay pestañas.
- **Estados persistentes:** mantiene el estado de las ventanas activas.

3.7.3. Clase LanguageItem

Representa un elemento de idioma en el sistema de internacionalización.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/LanguageItem.java

Funcionalidades:

- Encapsula información completa de un idioma (código, nombre, bandera).
- Proporciona objeto Locale para configuración regional.
- Implementa patrón Value Object con métodos de comparación.
- Integración con componentes de interfaz JavaFX.

3.8. Patrones de diseño implementados

3.8.1. Patrón MVC (Modelo-Vista-Controlador)

La aplicación implementa claramente el patrón arquitectónico Modelo-Vista-Controlador:

- **Modelo**: clases en el paquete `gramatica` (Gramatica, Simbolo, Terminal, NoTerminal).
- **Vista**: archivos FXML en el directorio `vistas` y estilos CSS.
- **Controlador**: clases como Editor, SimulacionFinal, MenuPrincipal que manejan la lógica de presentación.

Referencia: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Gramatica.java (modelo)

3.8.2. Patrón Observer

Utilizado extensivamente para la gestión de eventos de la interfaz de usuario:

- Observadores de JavaFX para cambios en propiedades.
- Sistema de internacionalización reactiva.
- Notificación automática de cambios de estado.

3.8.3. Patrón Factory Method

Implementado para la creación especializada de componentes:

- Creación de símbolos terminales y no terminales.
- Factorías de componentes de interfaz.
- Creación dinámica de pestañas y ventanas.

Referencia: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/TabManager.java

3.8.4. Patrón Strategy

Utilizado para algoritmos intercambiables:

- Diferentes estrategias de análisis sintáctico.
- Estrategias de manejo de errores.
- Algoritmos de transformación gramatical.

3.9. Consideraciones de rendimiento

3.9.1. Gestión de memoria

- Uso de `HashSet` para conjuntos PRIMERO y SIGUIENTE (acceso O(1)).
- Implementación eficiente de algoritmos de análisis sintáctico $O(n^3)$.
- Gestión adecuada de recursos JavaFX con limpieza automática.
- Cache inteligente de componentes FXML.

3.9.2. Optimizaciones implementadas

- Cálculo incremental de conjuntos PRIMERO y SIGUIENTE.
- Cache de resultados de análisis sintáctico.
- Lazy loading de componentes de interfaz.
- Optimización de renderizado JavaFX.

3.10. Tratamiento de errores

3.10.1. Validación de gramáticas

La aplicación implementa un sistema completo de validación:

- Verificación de gramáticas bien formadas.
- Detección automática de conflictos LL(1).
- Validación sintáctica de símbolos y producciones.
- Mensajes de error contextuales en múltiples idiomas.

Referencia: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Gramatica.java (método esLL1())

3.10.2. Manejo de excepciones

Sistema robusto de manejo de errores:

- Excepciones específicas para diferentes tipos de errores.
- Mensajes de error informativos para el usuario.
- Recuperación graceful con estrategias de fallback.
- Logging detallado para debugging.

3.10.3. Funciones de error personalizables

- Sistema de 7 funciones de error predefinidas.
- Creación de funciones de error personalizadas.
- Recuperación automática de errores durante simulación.
- Configuración de estrategias de manejo de errores.

Capítulo 4

Documentación de Paquetes

4.1. Introducción

Este capítulo proporciona una documentación detallada de cada paquete Java que constituye la aplicación SimAS 3.0. Para cada paquete se incluye una descripción de su propósito, las clases que contiene, las relaciones con otros paquetes y referencias directas al código fuente en el repositorio.

Referencias al Manual Técnico: Para explicaciones más detalladas sobre la arquitectura de paquetes, diagramas de dependencias completos, métricas de complejidad, análisis de acoplamiento/cohesión y patrones de diseño implementados, consulte:

- Capítulo 8: "Diseño de paquetes Arquitectura completa y análisis detallado de dependencias.
- Capítulo 9: "Diseño de clases Implementación detallada de todas las clases del sistema.

La arquitectura de paquetes sigue principios de diseño orientado a objetos con separación clara de responsabilidades, bajo acoplamiento y alta cohesión. El sistema está organizado en 8 paquetes principales distribuidos en capas funcionales.

4.2. Arquitectura general de paquetes

SimAS 3.0 está estructurado siguiendo una *arquitectura modular por capas* con separación clara de responsabilidades:

- **Capa de Presentación** (`bienvenida, vistas`): gestiona la interfaz de usuario.
- **Capa de Lógica de Negocio** (`editor, simulador`): implementa la funcionalidad principal.
- **Capa de Modelo de Datos** (`gramatica`): representa las estructuras de datos fundamentales

- **Capa de Servicios Transversales** (`utils`): proporciona funcionalidades comunes.
- **Capa de Recursos** (`resources`): contiene recursos estáticos.
- **Capa de Documentación** (`centroayuda`): proporciona ayuda y documentación.

Referencia a la arquitectura completa: Consulte el Capítulo 8 del Manual Técnico para análisis detallado de la arquitectura de paquetes.

4.3. Paquete bienvenida

4.3.1. Propósito

El paquete `bienvenida` implementa la *capa de presentación inicial* del sistema, actuando como punto de entrada principal de SimAS 3.0. Este paquete es fundamental para la experiencia del usuario, proporcionando una interfaz intuitiva y profesional.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/bienvenida

Responsabilidades principales:

- Gestión completa del ciclo de vida inicial de la aplicación.
- Navegación centralizada hacia módulos funcionales.
- Coordinación entre componentes de la interfaz principal.
- Gestión de estados globales de la aplicación.

4.3.2. Clases principales

4.3.2.1. Clase Bienvenida

Clase principal que gestiona la pantalla de bienvenida de la aplicación.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/bienvenida/Bienvenida.java

- **Herencia**: Extiende `Application` de JavaFX.
- **Patrón implementado**: Singleton para garantizar instancia única.
- **Responsabilidades**:
  - Mostrar la pantalla de bienvenida con información del proyecto.
  - Gestionar la transición automática al menú principal (2.5 segundos).
  - Configurar el estilo y comportamiento de la ventana de bienvenida.

- Soporte completo de internacionalización.

- **Métodos principales**:
  - `start(Stage)`: inicializa la ventana de bienvenida
  - `abrirMenuPrincipal()`: transita al menú principal
  - `main(String[])`: punto de entrada estático

4.3.2.2. Clase MenuPrincipal

Controlador principal que gestiona el menú principal de la aplicación y coordina la navegación.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/bienvenida/MenuPrincipal.java

- **Herencia**: extiende `Application` de JavaFX.
- **Patrón implementado**: Facade para simplificar acceso a módulos complejos.
- **Responsabilidades**:
  - Gestionar la interfaz del menú principal con sistema de pestañas avanzado.
  - Coordinar la apertura de diferentes módulos (editor, simulador, ayuda).
  - Implementar internacionalización completa con cambio dinámico de idioma.
  - Gestionar sistema de ventanas secundarias y arrastrar/soltar pestañas.
  - Configurar atajos de teclado para operaciones frecuentes.
  - Gestionar navegación contextual y estados de la aplicación.
- **Métodos principales**:
  - `start(Stage)`: inicializa la ventana principal completa.
  - `cambiarIdioma()`: gestiona cambio dinámico de idioma.
  - `onBtnEditorAction()`: abre nueva instancia del editor.
  - `onBtnSimuladorAction()`: carga gramática y abre simulador.
  - `onBtnAyudaAction()`: abre documentación en navegador.

4.3.3. Dependencias y relaciones

- **Dependencias críticas**:
  - `utils.*`: sistema de internacionalización, gestión de pestañas y ventanas (**85 % del sistema**).
  - `resources`: iconos y recursos gráficos para la interfaz.
  - `vistas`: archivos FXML para definición de interfaces.

- **Relaciones con otros paquetes**:
  - `editor`: crea instancias del editor de gramáticas.
  - `simulador`: coordina lanzamiento del simulador descendente.
  - `centroayuda`: integra sistema de ayuda contextual.
  - `gramatica`: utiliza modelo de datos para gestión de gramáticas.

Características técnicas:

- **Complejidad**: 2 clases (5 % del total del sistema).
- **Patrones implementados**: Singleton, Facade, MVC (Vista-Controlador).
- **Responsabilidad**: punto de entrada único y navegación centralizada.

4.4. Paquete editor

4.4.1. Propósito

El paquete `editor` implementa el *núcleo funcional de edición de gramáticas* de SimAS 3.0, proporcionando una interfaz completa para la creación, modificación y gestión de gramáticas de contexto libre. Este paquete representa el componente más complejo del sistema, con 10 clases que implementan un asistente paso a paso altamente sofisticado.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/editor

Arquitectura y diseño:

El paquete sigue una *arquitectura modular jerárquica* organizada en tres niveles:

1. **Nivel de Control Principal**: `Editor` y `EditorWindow` (gestión global).
2. **Nivel de Coordinación**: `PanelCreacionGramatica` (orquestación del asistente).
3. **Nivel de Especialización**: paneles específicos por funcionalidad.

4.4.2. Clases principales

4.4.2.1. Clase Editor

Clase principal del editor que coordina todos los paneles de edición y gestiona el estado global.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/editor/Editor.java

- **Herencia**: extiende `VBox` de JavaFX e implementa `ActualizableTextos`
- **Patrón implementado**: Mediator para coordinar comunicación entre paneles.
- **Responsabilidades**:
  - Gestionar la gramática activa y su estado.
  - Coordinar todos los paneles del asistente de creación.
  - Implementar patrón Mediator para comunicación entre componentes.
  - Gestionar persistencia de datos y validación global.
  - Coordinar navegación entre pasos del asistente.
  - Gestionar integración con el sistema de pestañas.
- **Métodos principales**:
  - `Editor(TabPane, MenuPrincipal)`: constructor con configuración completa.
  - `cargarGramatica(Gramatica)`: carga gramática para edición.
  - `validarGramaticaActual()`: validación completa de la gramática.
  - `guardarGramatica()`: persistencia de datos.

4.4.2.2. EditorWindow.java

Ventana principal del editor que contiene todos los paneles de edición.

- **Responsabilidades**:
  - Gestionar la interfaz de usuario del editor.
  - Coordinar la navegación entre paneles.
  - Gestionar el estado de la gramática en edición.

4.4.2.3. PanelCreacionGramatica.java

Panel base que define la estructura común para todos los paneles de creación.

4.4.2.4. PanelCreacionGramaticaPaso1.java

Panel para la definición del nombre y descripción de la gramática.

4.4.2.5. PanelCreacionGramaticaPaso2.java

Panel para la definición de símbolos terminales y no terminales de la gramática.

4.4.2.6. PanelCreacionGramaticaPaso3.java

Panel para la definición de producciones de la gramática.

4.4.2.7. PanelCreacionGramaticaPaso4.java

Panel para la definición del símbolo inicial y validación de la gramática.

4.4.2.8. PanelProducciones.java

Panel especializado para la gestión de producciones con funcionalidades avanzadas.

4.4.2.9. PanelSimbolosNoTerminales.java

Panel especializado para la gestión de símbolos no terminales.

4.4.2.10. PanelSimbolosTerminales.java

Panel especializado para la gestión de símbolos terminales.

4.4.3. Dependencias

- `gramatica.*`: para el modelo de datos de gramáticas.
- `utils.*`: para utilidades de interfaz y gestión de ventanas.
- `vistas.*`: para las definiciones FXML de la interfaz.

4.5. Paquete gramatica

4.5.1. Propósito

El paquete `gramatica` contiene el modelo de datos y los algoritmos fundamentales para el manejo de gramáticas libres de contexto y la generación de tablas de análisis predictivo.

4.5.2. Clases principales

4.5.2.1. Gramatica.java

Clase central que representa una gramática libre de contexto completa.

- **Responsabilidades**:
  - Almacenar todos los componentes de una gramática.
  - Implementar algoritmos de análisis sintáctico.
  - Generar tablas de análisis predictivo.
  - Validar la gramática y detectar conflictos.
- **Métodos principales**:
  - `generarTablaPredictiva()`: genera la tabla de análisis predictivo.
  - `calcularFirst()`: calcula los conjuntos PRIMERO.
  - `calcularFollow()`: calcula los conjuntos SIGUIENTE.
  - `esLL1()`: verifica si la gramática es LL(1).

4.5.2.2. Simbolo.java

Clase abstracta base para todos los símbolos de la gramática.

4.5.2.3. Terminal.java

Representa un símbolo terminal de la gramática.

4.5.2.4. NoTerminal.java

Representa un símbolo no terminal de la gramática.

4.5.2.5. Produccion.java

Representa una producción de la gramática con su antecedente y consecuente.

4.5.2.6. Antecedente.java

Representa el lado izquierdo de una producción (no terminal).

4.5.2.7. Consecuente.java

Representa el lado derecho de una producción (secuencia de símbolos).

4.5.2.8. TablaPredictiva.java

Clase base para la representación de tablas de análisis predictivo.

4.5.2.9. TablaPredictivaPaso5.java

Implementación específica de la tabla predictiva con funcionalidades avanzadas.

4.5.2.10. FilaTablaPredictiva.java

Representa una fila de la tabla predictiva.

4.5.2.11. FuncionError.java

Representa una función de error personalizada para el análisis.

4.5.3. Algoritmos implementados

4.5.3.1. Cálculo de conjuntos PRIMERO

El algoritmo calcula para cada símbolo no terminal el conjunto de símbolos terminales que pueden aparecer al inicio de las cadenas derivadas.

4.5.3.2. Cálculo de conjuntos SIGUIENTE

El algoritmo calcula para cada símbolo no terminal el conjunto de símbolos terminales que pueden aparecer inmediatamente después de él en alguna derivación.

4.5.3.3. Generación de tabla predictiva

Utiliza los conjuntos PRIMERO y SIGUIENTE para construir la tabla de análisis predictivo LL(1).

4.5.4. Dependencias

Este paquete es independiente y no tiene dependencias externas, sirviendo como base para otros paquetes.

4.6. Paquete simulador

4.6.1. Propósito

El paquete `simulador` implementa el simulador de análisis sintáctico descendente predictivo, permitiendo a los usuarios simular el proceso de análisis paso a paso.

4.6.2. Clases principales

4.6.2.1. PanelSimuladorDesc.java

Panel principal del simulador que coordina todos los componentes de simulación.

4.6.2.2. PanelSimulacion.java

Panel que gestiona la interfaz de usuario del simulador y el estado de la simulación.

4.6.2.3. SimulacionFinal.java

Clase que implementa la lógica de simulación del análisis sintáctico.

4.6.2.4. PanelNuevaSimDescPaso*.java

Conjunto de paneles para la configuración paso a paso de una nueva simulación.

4.6.2.5. EditorCadenaEntradaController.java

Controlador para la edición de cadenas de entrada a analizar.

4.6.2.6. NuevaFuncionError.java

Panel para la creación de nuevas funciones de error personalizadas.

4.6.2.7. PanelGramaticaOriginal.java

Panel que muestra la gramática original utilizada en la simulación.

4.6.3. Características del simulador

- Simulación paso a paso del análisis sintáctico.
- Visualización de la pila de análisis.
- Visualización del estado de la entrada.
- Generación del árbol de derivación.
- Manejo de errores con funciones personalizadas.
- Interfaz intuitiva con controles de navegación.

4.6.4. Dependencias

- `gramatica.*`: para acceso a gramáticas y tablas predictivas.
- `utils.*`: para utilidades de interfaz.
- `vistas.*`: para las definiciones FXML.

4.7. Paquete utils

4.7.1. Propósito

El paquete `utils` contiene utilidades generales, servicios de internacionalización y herramientas de gestión de ventanas que son utilizadas por toda la aplicación.

4.7.2. Clases principales

4.7.2.1. SecondaryWindow.java

Utilidad para la gestión de ventanas secundarias de la aplicación.

4.7.2.2. TabManager.java

Gestiona el sistema de pestañas para múltiples proyectos simultáneos.

4.7.2.3. TabPaneMonitor.java

Monitor que gestiona el estado y comportamiento de las pestañas.

4.7.2.4. ActualizableTextos.java

Interfaz para componentes que pueden actualizar sus textos según el idioma seleccionado.

4.7.2.5. LanguageItem.java

Representa un elemento de idioma en el sistema de internacionalización.

4.7.2.6. LanguageListCell.java

Celda personalizada para la visualización de idiomas en listas.

4.7.3. Sistema de internacionalización

El paquete incluye archivos de propiedades para múltiples idiomas:

- `messages_es.properties`: textos en español.
- `messages_en.properties`: textos en inglés.
- `messages_de.properties`: textos en alemán.
- `messages_fr.properties`: textos en francés.
- `messages_ja.properties`: textos en japonés.
- `messages_pt.properties`: textos en portugués.

4.7.4. Dependencias

Este paquete es independiente y proporciona servicios a otros paquetes.

4.8. Paquete centroayuda

4.8.1. Propósito

El paquete `centroayuda` implementa el sistema de ayuda integrado de la aplicación, incluyendo documentación, tutoriales y información sobre el proyecto.

4.8.2. Clases principales

4.8.2.1. AcercaDe.java

Ventana que muestra información sobre la aplicación, desarrolladores y versión.

4.8.3. Recursos incluidos

- `ayuda.html`: documentación principal de ayuda.
- `SimAS.html`: información específica sobre la aplicación.
- `Tema_*.pdf`: documentos temáticos sobre análisis sintáctico.
- `imagenes/`: recursos gráficos para la documentación.

4.8.4. Dependencias

- `utils.*`: para utilidades de interfaz.

4.9. Paquete vistas

4.9.1. Propósito

El paquete `vistas` contiene todos los archivos FXML que definen las interfaces de usuario de la aplicación, siguiendo el patrón de separación de vista y lógica.

4.9.2. Archivos FXML principales

- `Bienvenida.fxml`: pantalla de bienvenida.
- `MenuPrincipal.fxml`: menú principal de la aplicación.
- `Editor.fxml`: interfaz principal del editor.
- `PanelCreacionGramaticaPaso*.fxml`: paneles de creación de gramáticas.
- `PanelSimulacion.fxml`: interfaz del simulador.
- `SimulacionFinal.fxml`: interfaz de simulación final.
- `styles2.css`: estilos CSS para la aplicación.

4.9.3. Características de las vistas

- Diseño responsivo y adaptable.
- Uso de estilos CSS para personalización.
- Integración con el sistema de internacionalización.
- Separación clara entre presentación y lógica.

4.10. Métricas y análisis de la arquitectura

4.10.1. Distribución de clases por paquete

Con el conocimiento detallado de las 39 clases documentadas, podemos proporcionar métricas precisas sobre la complejidad del sistema:

Tabla 4.1: Distribución de clases Java por paquete

Paquete	**Número de clases**	**Porcentaje**
simulador	13	33 %
editor	10	26 %
gramatica	8	21 %
utils	6	15 %
bienvenida	2	5 %
Total	**39**	**100 %**

4.10.2. Matriz de dependencias completa

Tabla 4.2: Matriz de dependencias entre paquetes

De → A	bienv.	editor	sim.	gram.	utils	res.	centro	vistas
bienvenida	-	→	→	-	→	→	-	→
editor	-	-	-	→	→	→	→	→
simulador	-	-	-	→	→	→	→	→
gramatica	-	-	-	-	-	-	-	-
utils	-	-	-	-	-	-	-	-
resources	-	-	-	-	-	-	-	-
centroayuda	-	-	-	-	-	→	-	-
vistas	-	-	-	-	-	-	-	-

4.10.3. Complejidad algorítmica por paquete

Tabla 4.3: Complejidad algorítmica de los paquetes principales

Paquete	**Complejidad algorítmica**
gramatica	O(n) a O(n^3) (cálculo de conjuntos, construcción de tabla)
simulador	O(n^3) (algoritmos de análisis sintáctico)
editor	O(n^2) (validación y transformación de gramáticas)
utils	O(1) a O(n) (gestión de componentes y navegación)
bienvenida	O(1) (navegación básica)

4.10.4. Análisis de acoplamiento y cohesión

- **Acoplamiento bajo**: las dependencias están claramente definidas y minimizadas.
- **Cohesión alta**: las clases dentro de cada paquete comparten fuerte relación funcional.
- **Punto único de fallo**: el paquete `gramatica` es crítico para todo el sistema.
- **Servicios transversales**: los paquetes `utils` y `resources` son utilizados por el 85 % del sistema.

Referencias al repositorio:

- **Código fuente completo**: https://github.com/Llamatekee/SimAS-3.0/tree/main/src
- **Recursos del sistema**: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/resources
- **Archivos de interfaz**: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/vistas

- **Documentación técnica**: consulte Capítulo 8 del Manual Técnico para análisis detallado.

Capítulo 5

Sistema de Internacionalización

5.1. Introducción

SimAS 3.0 implementa un sistema completo de internacionalización (i18n) que permite a la aplicación adaptarse a diferentes idiomas y regiones. Este sistema soporta 6 idiomas completos con conmutación en tiempo de ejecución, facilitando el acceso a usuarios de diferentes países y culturas.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/utils

Este capítulo documenta la arquitectura, implementación y uso del sistema de internacionalización, con referencias directas al código fuente en el repositorio.

Referencias al Manual Técnico: para explicaciones más detalladas sobre las clases, paquetes y dependencias del sistema de internacionalización, incluyendo diagramas de clases, análisis de complejidad y patrones de diseño implementados, consulte:

- Capítulo 8: "Diseño de paquetes Arquitectura general y dependencias.
- Capítulo 9: "Diseño de clases Implementación detallada de clases como LanguageItem y LanguageListCell.

5.2. Arquitectura del sistema

5.2.1. Componentes principales

El sistema de internacionalización está compuesto por los siguientes elementos:

- **Archivos de propiedades**: contienen las traducciones para cada idioma (ubicados en `src/utils/`).
- **LanguageItem**: representa un idioma disponible con su configuración completa.

- **LanguageListCell**: celda personalizada para mostrar idiomas en interfaces.
- **ActualizableTextos**: interfaz para componentes que pueden actualizar sus textos.
- **ResourceBundle**: mecanismo de Java para cargar recursos localizados.
- **Internacionalización reactiva**: sistema de actualización automática de textos.

5.2.2. Idiomas soportados

La aplicación soporta los siguientes idiomas:

Código	Idioma	Archivo
es	Español	messages_es.properties
en	English	messages_en.properties
de	Deutsch	messages_de.properties
fr	Français	messages_fr.properties
ja	Japanese	messages_ja.properties
pt	Português	messages_pt.properties

Tabla 5.1: Idiomas soportados en SimAS 3.0

5.3. Implementación

5.3.1. Clase LanguageItem

La clase `LanguageItem` representa un idioma disponible en el sistema de internacionalización.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/LanguageItem.java

Características principales:

- **name**: nombre del idioma en su idioma nativo.
- **code**: código ISO del idioma.
- **flag**: bandera representativa del país/región.
- **locale**: objeto Locale de Java para la localización.

Referencia: Para una explicación más detallada de la clase LanguageItem, incluyendo su implementación completa, atributos, métodos y relaciones con otras clases, consulte el Capítulo 9 del Manual Técnico ("Diseño de clases").

5.3.2. Clase LanguageListCell

Celda personalizada para mostrar idiomas en listas desplegables del sistema de internacionalización.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/LanguageListCell.java

Funcionalidades:

- Muestra la bandera del país/región.
- Muestra el nombre del idioma.
- Formato visual atractivo.
- Integración con JavaFX.

Referencia: Para una explicación más detallada de la clase LanguageListCell, incluyendo su implementación completa, métodos de renderizado y integración con componentes JavaFX, consulte el Capítulo 9 del Manual Técnico ("Diseño de clases").

5.3.3. Interfaz ActualizableTextos

Interfaz fundamental que permite a los componentes actualizar sus textos dinámicamente cuando cambia el idioma.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/ActualizableTextos.java

Funcionalidades:

- Define contrato para componentes actualizables.
- Método `actualizarTextos(ResourceBundle)` para actualización de textos.
- Implementada por 25+ componentes en toda la aplicación.
- Soporte para actualización automática y manual.

5.4. Archivos de propiedades

5.4.1. Estructura de los archivos

Los archivos de propiedades siguen el formato estándar de Java y contienen todas las traducciones organizadas por secciones funcionales.

Estructura de archivos:

- **Idioma base (español)**: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/messages_es.properties
- **Inglés**: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/messages_en.properties
- **Alemán**: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/messages_de.properties
- **Francés**: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/messages_fr.properties
- **Japonés**: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/messages_ja.properties
- **Portugués**: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/messages_pt.properties

Organización del contenido:

Cada archivo contiene aproximadamente 150+ claves organizadas en secciones:

- **Interfaz principal**: menús, botones, títulos.
- **Editor de gramáticas**: pasos del asistente, validaciones.
- **Simulador**: estados de simulación, acciones, errores.
- **Mensajes del sistema**: errores, confirmaciones, notificaciones.
- **Ayuda contextual**: tooltips y mensajes de ayuda.

Características técnicas:

- Codificación UTF-8 para soporte completo de caracteres internacionales.
- Formato clave=valor estándar de Java.
- Comentarios descriptivos para facilitar mantenimiento.
- Validación automática de integridad de traducciones.

5.5. Integración con la interfaz

5.5.1. Cambio dinámico de idioma

El sistema permite cambiar el idioma de la aplicación en tiempo de ejecución:

```
public void cambiarIdioma(LanguageItem idiomaSeleccionado) {
    currentLocale = idiomaSeleccionado.getLocale();
    bundle = ResourceBundle.getBundle("utils.messages", currentLocale);

    // Actualizar textos de la interfaz
    actualizarTextos();

    // Notificar a otros componentes
    notificarCambioIdioma();
}
```

Listing 5.1: Método para cambiar idioma

5.5.2. Actualización de textos

Los componentes que implementan `ActualizableTextos` pueden actualizar sus textos:

```
@Override
public void actualizarTextos(ResourceBundle bundle) {
    labelTitulo.setText(bundle.getString("menu.title"));
    labelSubtitulo.setText(bundle.getString("menu.subtitle"));
    btnEditor.setText(bundle.getString("menu.editor"));
    btnSimulador.setText(bundle.getString("menu.simulator"));
    btnAyuda.setText(bundle.getString("menu.help"));
    btnSalir.setText(bundle.getString("menu.exit"));
}
```

Listing 5.2: Implementación de actualización de textos

5.6. Configuración y uso

5.6.1. Inicialización del sistema

El sistema se inicializa al arrancar la aplicación:

```
private void inicializarInternacionalizacion() {
    // Cargar idioma por defecto (español)
    currentLocale = new Locale("es");
    bundle = ResourceBundle.getBundle("utils.messages", currentLocale);

    // Configurar combo de idiomas
    configurarComboIdiomas();

    // Actualizar textos iniciales
    actualizarTextos();
}
```

Listing 5.3: Inicialización del sistema de i18n

5.6.2. Configuración del selector de idiomas

```
1  private void configurarComboIdiomas() {
2      List<LanguageItem> idiomas = Arrays.asList(
3          new LanguageItem("Espanol", "es", "espana.png"),
4          new LanguageItem("English", "en", "england.png"),
5          new LanguageItem("Deutsch", "de", "alemania.png"),
6          new LanguageItem("Francais", "fr", "francia.png"),
7          new LanguageItem("Japanese", "ja", "japon.png"),
8          new LanguageItem("Portugues", "pt", "portugal.png")
9      );
10
11     comboIdioma.setItems(FXCollections.observableArrayList(idiomas));
12     comboIdioma.setCellFactory(listView -> new LanguageListCell());
13     comboIdioma.setButtonCell(new LanguageListCell());
14
15     // Seleccionar idioma actual
16     comboIdioma.getSelectionModel().select(
17         idiomas.stream()
18                 .filter(idioma ->
    idioma.getCode().equals(currentLocale.getLanguage()))
19                 .findFirst()
20                 .orElse(idiomas.get(0))
21     );
22 }
```

Listing 5.4: Configuración del selector de idiomas

5.7. Recursos gráficos

5.7.1. Banderas de países

El sistema utiliza imágenes de banderas para representar visualmente cada idioma:

- `espana.png`: bandera de España.
- `england.png`: bandera de Inglaterra.
- `alemania.png`: bandera de Alemania.
- `francia.png`: bandera de Francia.
- `japon.png`: bandera de Japón.
- `portugal.png`: bandera de Portugal.

5.7.2. Ubicación de recursos

Las banderas se encuentran en el directorio `src/resources/` y se cargan como recursos de la aplicación.

5.8. Consideraciones técnicas

5.8.1. Rendimiento

- Los archivos de propiedades se cargan una sola vez al inicializar.
- El cambio de idioma es instantáneo.
- No hay impacto significativo en el rendimiento.

5.8.2. Mantenimiento

- Fácil adición de nuevos idiomas.
- Estructura clara y organizada.
- Separación entre código y textos.

5.8.3. Extensibilidad

El sistema está diseñado para ser fácilmente extensible:

- Agregar nuevos idiomas solo requiere crear un archivo de propiedades.
- Agregar nuevos textos requiere actualizar todos los archivos de propiedades.
- La interfaz se adapta automáticamente a nuevos idiomas.

5.9. Mejores prácticas implementadas

5.9.1. Separación de contenido y código

- Todos los textos visibles están en archivos de propiedades.
- No hay cadenas hardcodeadas en el código.
- Fácil localización por parte de traductores.

5.9.2. Gestión de recursos

- Uso eficiente de ResourceBundle.
- Carga lazy de recursos gráficos.
- Gestión adecuada de memoria.

5.9.3. Experiencia de usuario

- Cambio de idioma inmediato.
- Interfaz intuitiva para selección de idioma.
- Persistencia de la preferencia de idioma.

5.10. Métricas del sistema de internacionalización

5.10.1. Cobertura de internacionalización

- **6 idiomas completamente soportados**: Español, Inglés, Alemán, Francés, Japonés, Portugués.
- **150+ claves de traducción** por archivo de propiedades.
- **25+ componentes actualizables** que implementan `ActualizableTextos`.
- **100 % de cobertura**: todos los textos de interfaz están internacionalizados.

5.10.2. Implementación técnica

- **ResourceBundle estándar**: uso de la API nativa de Java para internacionalización.
- **UTF-8 completo**: soporte para caracteres internacionales y símbolos especiales.
- **Cambio dinámico**: conmutación de idioma en tiempo de ejecución sin reinicio.
- **Persistencia**: mantenimiento de la selección de idioma entre sesiones.

5.11. Pruebas y validación

5.11.1. Verificación de traducciones

- Validación automática de completitud de traducciones.
- Verificación de formato y sintaxis de archivos de propiedades.
- Pruebas de carga y acceso a recursos localizados.
- Detección automática de claves faltantes o inconsistentes.

5.11.2. Pruebas de interfaz

- Verificación de adaptación automática de textos largos.
- Pruebas exhaustivas de cambio dinámico de idioma.
- Validación de recursos gráficos y banderas de países.
- Testing de usabilidad en diferentes idiomas.

5.11.3. Herramientas de desarrollo

- Scripts de validación automática de traducciones.
- Herramientas de comparación entre archivos de idiomas.
- Sistema de logging para debugging de internacionalización.

Referencias al código fuente:

- **Framework de i18n**: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/utils
- **Archivos de propiedades**: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/utils (`messages_*.properties`)
- **Recursos gráficos**: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/resources

Capítulo 6

Compilación y Despliegue

6.1. Introducción

Este capítulo describe el proceso completo de compilación, empaquetado y despliegue de la aplicación SimAS 3.0. Se incluyen referencias a los scripts de construcción en el repositorio, configuración de dependencias y procedimientos para generar ejecutables nativos multiplataforma.

Referencias al Manual Técnico: Para información detallada sobre la arquitectura del sistema de construcción, configuración de entornos de desarrollo y análisis de rendimiento del sistema, consulte:

- Capítulo 8: "Diseño de paquetes Estructura del sistema y dependencias.
- Capítulo 9: "Diseño de clases Clases relacionadas con configuración y utilidades del sistema.

Ubicación en el repositorio: https://github.com/Llamatekee/SimAS-3.0/tree/main

6.2. Herramientas de construcción

6.2.1. Java Development Kit (JDK)

Requisitos del sistema:

- JDK 17 o superior (OpenJDK o Oracle JDK).
- Herramienta `jpackage` (incluida desde JDK 14+).
- Variables de entorno `JAVA_HOME` y `PATH` configuradas.
- JavaFX SDK 17.0.12 (no incluido en JDK estándar).

Verificación de instalación:

- `java -version` - Verificar versión de Java.
- `javac -version` - Verificar compilador Java.
- `jpackage -version` - Verificar herramienta de empaquetado.

Configuración recomendada:

```
export JAVA_HOME=/path/to/jdk17
export PATH=$JAVA_HOME/bin:$PATH
export JAVAFX_HOME=/path/to/javafx-sdk-17.0.12
```

Listing 6.1: Configuración de variables de entorno

6.2.2. JavaFX SDK

Versión utilizada: JavaFX 17.0.12

Ubicación: `lib/javafx-sdk-17.0.12/`

Componentes incluidos:

- `lib/`: Librerías JavaFX.
- `bin/`: Herramientas de JavaFX.
- `legal/`: Licencias y documentación legal.

6.3. Scripts de construcción

6.3.1. Script build.sh (Unix/Linux/macOS)

Script principal para compilación en sistemas Unix-like.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/build.sh

Funcionalidades principales:

- Configuración automática de variables de entorno.
- Verificación de dependencias (JDK, JavaFX).
- Creación de estructura de directorios `build/`.
- Compilación completa del código fuente Java.
- Copia de recursos (FXML, CSS, imágenes, internacionalización).

- Copia de librerías JavaFX al directorio de construcción.
- Generación del archivo JAR ejecutable con manifest.
- Optimización y minificación del paquete final.

6.3.2. Script build.bat (Windows)

Script equivalente para compilación en sistemas Windows.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/build.bat

Características específicas:

- Sintaxis de comandos nativa de Windows (`cmd`).
- Manejo de variables de entorno de Windows.
- Rutas con separadores de directorio de Windows (\).
- Integración con herramientas de desarrollo Windows.
- Soporte para PowerShell y Command Prompt.

6.3.3. Script create-standalone-app.sh

Script especializado para crear aplicaciones independientes.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/create-standalone-app.sh

Características principales:

- Utiliza `jpackage` para crear aplicaciones nativas.
- Incluye JRE embebido (no requiere Java instalado).
- Empaqueta todas las dependencias JavaFX.
- Genera ejecutables nativos (.app, .exe, .deb).
- Configuración específica por plataforma.
- Optimización de tamaño y rendimiento.

Resultado: Aplicación completamente autónoma en `dist-standalone/`

6.4. Proceso de compilación

6.4.1. Paso 1: Preparación del entorno

1. Clonar el repositorio:

```
    git clone https://github.com/Llamatekee/SimAS-3.0.git
    cd SimAS-3.0

```

2. Verificar instalación de JDK 17+ y JavaFX SDK

3. Configurar variables de entorno (si no están configuradas globalmente)

4. Dar permisos de ejecución a scripts:

```
    chmod +x build.sh
    chmod +x create-standalone-app.sh

```

6.4.2. Paso 2: Compilación del código fuente

- **Sistemas Unix/Linux/macOS:**

```
    ./build.sh

```

- **Sistemas Windows:**

```
    build.bat

```

Proceso interno: El script automatiza la compilación completa incluyendo verificación de dependencias, creación de directorios, compilación Java, copia de recursos y generación del JAR.

6.4.3. Paso 3: Estructura de archivos generada

Después de la compilación, se genera la siguiente estructura:

```
build/
|-- SimAS.jar                    # Archivo JAR principal
|-- lib/                         # Librerías JavaFX
|   |-- javafx.controls.jar
|   |-- javafx.fxml.jar
|   |-- ...
|-- resources/                   # Recursos de la aplicación
|   |-- logo.png
|   |-- icons/
|   |-- ...
|-- vistas/                      # Archivos FXML
```

```
|   |-- MenuPrincipal.fxml
|   |-- Bienvenida.fxml
|   |-- ...
|-- utils/                          # Archivos de propiedades
|   |-- messages_es.properties
|   |-- messages_en.properties
|   |-- ...
|-- MANIFEST.MF                     # Manifest del JAR
```

Listing 6.2: Estructura del directorio build

6.5. Creación de ejecutables nativos

6.5.1. Usando jpackage

Comando básico:

```
jpackage --input build \
         --main-jar SimAS.jar \
         --main-class bienvenida.Bienvenida \
         --name SimAS \
         --app-version 3.0 \
         --vendor "Antonio Llamas García" \
         --description "Simulador de Análisis Sintáctico" \
         --dest dist
```

Listing 6.3: Creación de ejecutable con jpackage

Parámetros específicos por plataforma:

6.5.1.1. macOS

```
jpackage --input build \
         --main-jar SimAS.jar \
         --main-class bienvenida.Bienvenida \
         --name SimAS \
         --type dmg \
         --app-version 3.0 \
         --mac-package-identifier com.simas.app \
         --mac-package-name "SimAS 3.0" \
         --dest dist
```

Listing 6.4: Configuración para macOS

6.5.1.2. Windows

```
jpackage --input build \
         --main-jar SimAS.jar \
         --main-class bienvenida.Bienvenida \
         --name SimAS \
         --type msi \
```

```
6           --app-version 3.0 \
7           --win-dir-chooser \
8           --win-menu \
9           --win-shortcut \
10          --dest dist
```

Listing 6.5: Configuración para Windows

6.5.1.3. Linux

```
1  jpackage --input build \
2           --main-jar SimAS.jar \
3           --main-class bienvenida.Bienvenida \
4           --name SimAS \
5           --type deb \
6           --app-version 3.0 \
7           --linux-shortcut \
8           --dest dist
```

Listing 6.6: Configuración para Linux

6.5.2. Aplicación independiente para macOS

El script `create-standalone-app.sh` crea una aplicación completamente independiente:

```
1  ./create-standalone-app.sh
```

Listing 6.7: Creación de aplicación independiente

Resultado:

- Archivo `SimAS.app` en `dist-standalone/`.
- Aplicación completamente autónoma.
- No requiere Java instalado en el sistema.
- Incluye todas las dependencias necesarias.

6.6. Configuración del manifest

6.6.1. Archivo MANIFEST.MF

El archivo manifest contiene metadatos importantes sobre la aplicación.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/build/MANIFEST.MF

Atributos principales:

- `Main-Class`: `bienvenida.Bienvenida` - Clase principal de la aplicación.
- `Class-Path`: rutas a las librerías JavaFX embebidas.
- `Implementation-Title`: nombre completo de la aplicación.
- `Implementation-Version`: versión actual (3.0).
- `Implementation-Vendor`: información del desarrollador.
- `Specification-Version`: versión de especificación.

Funciones del manifest:

- Define el punto de entrada de la aplicación.
- Especifica dependencias de classpath.
- Proporciona metadatos para herramientas de desarrollo.
- Habilita ejecución con `java -jar SimAS.jar`.

6.7. Distribución

6.7.1. Requisitos para usuarios finales

Opción 1: JAR ejecutable

- Java Runtime Environment (JRE) 17+.
- JavaFX Runtime (incluido en JRE 11+ o por separado).

Opción 2: Aplicación nativa

- No requiere Java instalado.
- Ejecutable nativo del sistema operativo.
- Mayor tamaño de descarga.

6.7.2. Instrucciones de instalación

6.7.2.1. Para desarrolladores

1. Clonar el repositorio.
2. Ejecutar `./build.sh` o `build.bat`.
3. Ejecutar `java -jar build/SimAS.jar`.

6.7.2.2. Para usuarios finales

1. Descargar la aplicación desde el repositorio.
2. Ejecutar `./dist-standalone/SimAS.app` (macOS).
3. O ejecutar `java -jar SimAS.jar` (requiere Java).

6.8. Solución de problemas

6.8.1. Errores comunes

6.8.1.1. Error: `JavaFX runtime components are missing`

Causa: JavaFX no está disponible en el classpath. **Solución:** asegurar que las librerías JavaFX estén en `lib/`.

6.8.1.2. Error: `Main class not found`

Causa: la clase principal no está especificada correctamente. **Solución:** verificar el manifest y la estructura de paquetes.

6.8.1.3. Error: `Permission denied`

Causa: scripts sin permisos de ejecución. **Solución:** `chmod +x build.sh create-standalone-ap`

6.8.2. Verificación de la instalación

- **Ejecutar la aplicación:**

```
1     java -jar build/SimAS.jar
2
```

- **Verificar estructura de archivos:**

```
1     ls -la build/
2
```

- **Verificar manifest:**

```
1     unzip -p build/SimAS.jar META-INF/MANIFEST.MF
2
```

6.9. Optimizaciones de rendimiento

6.9.1. Configuración de JVM

Parámetros recomendados para ejecución:

```
1 java -Xmx512m -Xms256m -jar SimAS.jar
```

Listing 6.8: Parámetros JVM optimizados

Explicación de parámetros:

- `-Xmx512m`: máximo heap de 512MB.
- `-Xms256m`: heap inicial de 256MB.
- `-jar`: ejecutar archivo JAR.

6.9.2. Reducción del tamaño

- Uso de ProGuard para ofuscación y minificación del código.
- Eliminación de dependencias no utilizadas con herramientas de análisis.
- Compresión de recursos (imágenes, archivos de configuración).
- Optimización de archivos JAR con `pack200`.

6.10. Automatización con CI/CD

6.10.1. GitHub Actions

El repositorio incluye configuración para compilación automática mediante GitHub Actions.

Ubicación del workflow: https://github.com/Llamatekee/SimAS-3.0/tree/main/.github/workflows

Características del pipeline CI/CD:

- **Compilación multiplataforma:** Linux, macOS y Windows.
- **Generación de releases:** generación automática de ejecutables.
- **Despliegue continuo:** publicación automática de versiones.

6.10.2. Beneficios de la automatización

- Verificación automática de cambios en cada commit.
- Compilación consistente en múltiples plataformas.
- Detección temprana de errores de compilación.
- Distribución simplificada de versiones.
- Historial completo de construcción.

6.11. Referencias al código fuente

- **Scripts de construcción:** https://github.com/Llamatekee/SimAS-3.0/tree/main
  - `build.sh` - Compilación Unix/Linux/macOS.
  - `build.bat` - Compilación Windows.
  - `create-standalone-app.sh` - Aplicación independiente.
- **Configuración de dependencias:** https://github.com/Llamatekee/SimAS-3.0/tree/main/lib
- **Ejecutables generados:** https://github.com/Llamatekee/SimAS-3.0/tree/main/dist-standalone
- **Documentación CI/CD:** https://github.com/Llamatekee/SimAS-3.0/tree/main/.github

6.12. Conclusión

El sistema de compilación y despliegue de SimAS 3.0 proporciona una solución completa y multiplataforma para el desarrollo y distribución de la aplicación. Los scripts automatizados facilitan el proceso de construcción, mientras que el uso de jpackage permite generar ejecutables nativos independientes.

Ventajas del sistema implementado:

- **Multiplataforma:** soporte completo para Windows, macOS y Linux.
- **Automatización:** scripts que automatizan todo el proceso de construcción.
- **Independencia:** aplicaciones que no requieren Java instalado.
- **Calidad:** verificación automática de dependencias y configuración.
- **Mantenibilidad:** código bien estructurado y documentado.

6.13. Consideraciones de seguridad

6.13.1. Firma de código

Para distribuir la aplicación de forma segura:

- Firmar el JAR con certificado digital.
- Firmar ejecutables nativos.
- Verificar integridad de archivos.

6.13.2. Verificación de dependencias

- Verificar checksums de librerías JavaFX.
- Actualizar dependencias regularmente.
- Usar versiones estables y soportadas.

Capítulo 7

Estructura del repositorio

7.1. Introducción

Este capítulo proporciona una guía completa para navegar y comprender la estructura del repositorio de SimAS 3.0. El repositorio está organizado de manera lógica y modular, facilitando la comprensión del código fuente y el mantenimiento del proyecto.

Ubicación del repositorio: https://github.com/Llamatekee/SimAS-3.0

Referencias al Manual Técnico: Para información detallada sobre la arquitectura del sistema, diagramas de clases y análisis de dependencias, consulte:

- Capítulo 8: "Diseño de paquetes Arquitectura completa.
- Capítulo 9: "Diseño de clases Implementación detallada.

7.2. Estructura general del repositorio

La estructura del repositorio sigue una organización clara y lógica:

```
SimAS-3.0/
|-- src/                          # Directorio principal del codigo
    fuente
|-- lib/                          # Librerias externas (JavaFX SDK)
|-- build/                        # Scripts y archivos de construccion
|-- dist-standalone/              # Ejecutables independientes generados
|-- resources/                    # Recursos graficos y archivos
    estaticos
|-- vistas/                       # Interfaces de usuario FXML
|-- fonts/                        # Fuentes tipograficas personalizadas
|-- .idea/                        # Configuracion de IntelliJ IDEA
|-- .gitignore                    # Archivos ignorados por control de
    versiones
|-- build.sh                      # Script de construccion para
    Unix/Linux/macOS
|-- build.bat                     # Script de construccion para Windows
|-- create-standalone-app.sh      # Generador de aplicaciones
    independientes
```

```
14 |-- README.md                      # Documentacion principal del proyecto
15 `-- ManualDeUsuario.pdf            # Manual de usuario en formato PDF
```

Listing 7.1: Estructura general del repositorio

7.3. Directorio src/ - Código fuente

El directorio `src/` contiene todo el código fuente organizado por paquetes:

```
1  src/
2  |-- bienvenida/                    # Punto de entrada de la aplicacion
3  |   |-- Bienvenida.java           # Pantalla de bienvenida
4  |   `-- MenuPrincipal.java        # Menu principal de navegacion
5  |-- editor/                       # Editor de gramaticas
6  |   |-- Editor.java               # Controlador principal del editor
7  |   |-- EditorWindow.java         # Ventana del editor
8  |   |-- PanelCreacionGramatica.java # Panel del asistente de creacion
      de gramaticas
9  |   |-- PanelCreacionGramaticaPaso1.java # Paso 1 del asistente de
      creacion de gramaticas
10 |   |-- PanelCreacionGramaticaPaso2.java # Paso 2 del asistente de
      creacion de gramaticas
11 |   |-- PanelCreacionGramaticaPaso3.java # Paso 3 del asistente de
      creacion de gramaticas
12 |   |-- PanelCreacionGramaticaPaso4.java # Paso 4 del asistente de
      creacion de gramaticas
13 |   |-- PanelProducciones.java # Panel de producciones
14 |   |-- PanelSimbolosNoTerminales.java # Panel de simbolos no terminales
15 |   `-- PanelSimbolosTerminales.java # Panel de simbolos terminales
16 |-- gramatica/                    # Modelo de datos gramatical
17 |   |-- Gramatica.java            # Clase principal del modelo
18 |   |-- Simbolo.java              # Clase base para simbolos
19 |   |-- Terminal.java             # Simbolos terminales
20 |   |-- NoTerminal.java           # Simbolos no terminales
21 |   |-- Produccion.java           # Reglas de produccion
22 |   |-- Antecedente.java          # Lado izquierdo de producciones
23 |   |-- Consecuente.java          # Lado derecho de producciones
24 |   |-- TablaPredictiva.java      # Tabla de analisis predictivo
25 |   |-- TablaPredictivaPaso5.java # Tabla de analisis predictivo paso 5
26 |   |-- FilaTablaPredictiva.java # Fila de tabla de analisis predictivo
27 |   `-- FuncionError.java         # Funciones de error
28 |-- simulador/                    # Motor de simulacion
29 |   |-- PanelSimulacion.java      # Panel basico de simulacion
30 |   |-- SimulacionFinal.java      # Motor principal
31 |   |-- PanelNuevaSimDescPaso1.java # Paso 1 del asistente de nueva
      simulacion descendente
32 |   |-- PanelNuevaSimDescPaso2.java # Paso 2 del asistente de nueva
      simulacion descendente
33 |   |-- PanelNuevaSimDescPaso3.java # Paso 3 del asistente de nueva
      simulacion descendente
34 |   |-- PanelNuevaSimDescPaso4.java # Paso 4 del asistente de nueva
      simulacion descendente
35 |   |-- PanelNuevaSimDescPaso5.java # Paso 5 del asistente de nueva
      simulacion descendente
36 |   |-- PanelNuevaSimDescPaso6.java # Simulador principal
```

```
37 |   |-- PanelNuevaSimDescPaso.java # Interfaz de pasos de simulacion
   descendente
38 |   |-- PanelGramaticaOriginal.java # Panel de gramatica original
39 |   |-- NuevaFuncionError.java # Panel de nueva funcion de error
40 |   `-- EditorCadenaEntradaController.java # Controlador de editor de
   cadena de entrada
41 |-- utils/                      # Utilidades transversales
42 |   |-- ActualizableTextos.java  # Interfaz de internacionalizacion
43 |   |-- LanguageItem.java        # Representa idiomas
44 |   |-- LanguageListCell.java    # Celda de idiomas
45 |   |-- SecondaryWindow.java     # Ventanas secundarias
46 |   |-- TabManager.java          # Gestion de pestanas
47 |   |-- TabPaneMonitor.java      # Monitor de pestanas
48 |   |-- messages_es.properties   # Textos en espanol
49 |   |-- messages_en.properties   # Textos en ingles
50 |   |-- messages_de.properties   # Textos en aleman
51 |   |-- messages_fr.properties   # Textos en frances
52 |   |-- messages_ja.properties   # Textos en japones
53 |   `-- messages_pt.properties   # Textos en portugues
54 |-- centroayuda/                # Sistema de ayuda
55 |   |-- AcercaDe.java            # Informacion del sistema
56 |   |-- ayuda.html               # Documentacion principal
57 |   |-- SimAS.html               # Informacion especifica
58 |   |-- Tema_1.pdf               # Documentos tematicos
59 |   |-- Tema_2.pdf               # Documentos tematicos
60 |   |-- Tema_3.pdf               # Documentos tematicos
61 |   |-- Tema_4.pdf               # Documentos tematicos
62 |   |-- Tema_5.pdf               # Documentos tematicos
63 |   `-- imagenes/                # Recursos graficos
64 |-- resources/                  # Recursos graficos
65 |   |-- simas-logo.png           # Logo principal
66 |   |-- simas-icon.png           # Icono de aplicacion
67 |   |-- uco-logo.png             # Logo institucional
68 |   |-- icons/                   # Iconos de acciones
69 |   |   |-- abrir.png
70 |   |   |-- guardar.png
71 |   |   |-- nuevo.png
72 |   |   |-- eliminar.png
73 |   |   |-- editar.png
74 |   |   `-- ...
75 |   `-- icons/                   # Mas iconos
76 |-- vistas/                     # Interfaces FXML
77 |   |-- Bienvenida.fxml          # Pantalla de bienvenida
78 |   |-- MenuPrincipal.fxml       # Menu principal
79 |   |-- Editor.fxml              # Editor de gramaticas
80 |   |-- PanelCreacionGramaticaPaso1.fxml # Paso 1 del asistente de
   creacion de gramaticas
81 |   |-- PanelCreacionGramaticaPaso2.fxml # Paso 2 del asistente de
   creacion de gramaticas
82 |   |-- PanelCreacionGramaticaPaso3.fxml # Paso 3 del asistente de
   creacion de gramaticas
83 |   |-- PanelCreacionGramaticaPaso4.fxml # Paso 4 del asistente de
   creacion de gramaticas
84 |   |-- PanelProducciones.fxml # Panel de producciones
85 |   |-- PanelSimbolosNoTerminales.fxml # Panel de simbolos no terminales
86 |   |-- PanelSimbolosTerminales.fxml # Panel de simbolos terminales
87 |   |-- PanelSimulacion.fxml # Panel de simulacion
88 |   |-- SimulacionFinal.fxml # Simulador principal
```

```
89 |   |-- EditorCadenaEntradaController.fxml # Controlador de editor de
       cadena de entrada
90 |   |-- NuevaFuncionError.fxml # Panel de nueva funcion de error
91 |   |-- VentanaGramaticaOriginal.fxml # Ventana de gramatica original
92 |   `-- styles2.css              # Estilos CSS
93 `-- centroayuda/                # Sistema de ayuda (duplicado)
```

Listing 7.2: Estructura del código fuente

7.4. Navegación por paquetes principales

7.4.1. Paquete bienvenida

Ubicación: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/bienvenida

Propósito: gestiona el punto de entrada y navegación inicial de la aplicación.

Clases principales:

- `Bienvenida.java` - Pantalla de bienvenida con transición automática.
- `MenuPrincipal.java` - Controlador principal de navegación.

Dependencias: paquetes utils, resources, vistas.

7.4.2. Paquete editor

Ubicación: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/editor

Propósito: implementa el asistente completo para creación y edición de gramáticas.

Clases principales:

- `Editor.java` - Controlador principal del editor.
- `PanelCreacionGramatica.java` - Orquestador del asistente.
- `PanelCreacionGramaticaPaso[1-4].java` - Pasos del asistente de creación de gramáticas.

Dependencias: paquetes gramatica, utils, vistas, resources.

7.4.3. Paquete gramatica

Ubicación: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/gramatica

Propósito: contiene el modelo de datos y algoritmos fundamentales del sistema.

Clases principales:

- `Gramatica.java` - Modelo principal de gramáticas.
- `Simbolo.java` - Jerarquía de símbolos.
- `TablaPredictiva.java` - Tablas de análisis predictivo.

Dependencias: ninguna (paquete base independiente).

7.4.4. Paquete simulador

Ubicación: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/simulador

Propósito: implementa el motor de simulación de análisis sintáctico.

Clases principales:

- `SimulacionFinal.java` - Motor principal de simulación.
- `PanelSimulacion.java` - Panel básico de simulación.
- `PanelNuevaSimDescPaso[1-6].java` - Pasos de configuración de la nueva simulación descendente.

Dependencias: paquetes gramatica, utils, vistas, resources.

7.4.5. Paquete utils

Ubicación: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/utils

Propósito: contiene utilidades transversales y servicios comunes.

Clases principales:

- `TabManager.java` - Gestión de pestañas jerárquicas.
- `SecondaryWindow.java` - Sistema de ventanas secundarias.
- `ActualizableTextos.java` - Framework de internacionalización.
- `LanguageItem.java` - Modelo de idiomas.

Dependencias: ninguna (servicios transversales).

7.5. Recursos y configuración

7.5.1. Directorio resources/

Ubicación: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/resources

Contiene todos los recursos gráficos de la aplicación:

- Logos e iconos de la aplicación.
- Banderas de países para internacionalización.
- Iconos de acciones y navegación.
- Recursos gráficos para documentación.

7.5.2. Directorio vistas/

Ubicación: https://github.com/Llamatekee/SimAS-3.0/tree/main/src/vistas

Contiene las definiciones de interfaces de usuario:

- Archivos FXML para cada ventana y panel.
- Hoja de estilos CSS principal (`styles2.css`).
- Componentes reutilizables de interfaz.

7.5.3. Archivos de configuración

Los archivos de configuración se encuentran en el directorio raíz:

- `.gitignore` - Archivos ignorados por Git.
- `.idea/` - Configuración de IntelliJ IDEA.
- Scripts de construcción (`build.sh`, `build.bat`).

7.6. Convenciones de nomenclatura

7.6.1. Nombres de paquetes

Los paquetes siguen la convención estándar de Java:

- Nombres en minúsculas.
- Palabras separadas por guiones bajos (solo cuando necesario).
- Nombres descriptivos del propósito del paquete.

7.6.2. Nombres de clases

Las clases siguen las convenciones estándar de Java:

- **PascalCase** para nombres de clases.
- Nombres descriptivos que indican la responsabilidad.
- Sufijos para tipos específicos (`Controller`, `Panel`, etc.).

7.6.3. Nombres de archivos

- Archivos Java: `NombreClase.java`.
- Archivos FXML: `NombreVentana.fxml`.
- Archivos de propiedades: `messages_xx.properties`.
- Scripts: `nombre-script.sh` o `nombre-script.bat`.

7.7. Navegación eficiente del código

7.7.1. Desde el repositorio GitHub

1. Acceder a https://github.com/Llamatekee/SimAS-3.0
2. Usar la búsqueda (`Ctrl+K`) para encontrar clases específicas.
3. Navegar por directorios usando el explorador de archivos.
4. Ver el historial de cambios de archivos específicos.

7.7.2. Desde el código local

1. Clonar el repositorio: `git clone https://github.com/Llamatekee/SimAS-3.0.git`
2. Abrir en IDE (IntelliJ IDEA, Eclipse, VS Code).
3. Usar la funcionalidad de búsqueda del IDE.
4. Explorar la jerarquía de paquetes en el navegador de proyectos.

7.7.3. Referencias cruzadas importantes

- **Punto de entrada:** `src/bienvenida/Bienvenida.java`.
- **Modelo principal:** `src/gramatica/Gramatica.java`.
- **Controlador principal:** `src/bienvenida/MenuPrincipal.java`.
- **Motor de simulación:** `src/simulador/SimulacionFinal.java`.
- **Framework i18n:** `src/utils/ActualizableTextos.java`.

7.8. Mantenimiento y evolución

7.8.1. Estructura extensible

La organización modular facilita:

- Adición de nuevos paquetes sin afectar existentes.
- Modificación de clases dentro de paquetes sin impacto global.
- Reutilización de componentes entre módulos.

7.8.2. Control de versiones

- `.gitignore` optimizado para proyectos Java/JavaFX.
- Historial completo de cambios en Git.
- Ramas para desarrollo de nuevas funcionalidades.
- Tags para versiones estables del sistema.

Esta estructura proporciona una base sólida para el mantenimiento, evolución y comprensión del código fuente de SimAS 3.0.

Capítulo 8

Convenciones de código y mejores prácticas

8.1. Introducción

Este capítulo documenta las convenciones de código, patrones de diseño y mejores prácticas implementadas en SimAS 3.0. Estas directrices garantizan la calidad, mantenibilidad y extensibilidad del código fuente.

Referencias al Manual Técnico: Para análisis detallado de patrones de diseño específicos y métricas de calidad del código, consulte:

- Capítulo 8: "Diseño de paquetes Patrones arquitectónicos.
- Capítulo 9: "Diseño de clases Patrones de diseño por clase.

8.2. Convenciones de nomenclatura

8.2.1. Nombres de paquetes

- **Convención:** nombres en minúsculas, sin guiones bajos.
- **Ejemplos:**
  - `bienvenida` - Punto de entrada.
  - `gramatica` - Modelo de datos.
  - `simulador` - Motor de simulación.
  - `utils` - Utilidades transversales.
- **Justificación:** sigue las convenciones estándar de Java y facilita la navegación.

8.2.2. Nombres de clases

- **Convención:** PascalCase (UpperCamelCase).
- **Ejemplos:**
  - `Gramatica` - Clase principal del modelo.
  - `SimulacionFinal` - Motor de simulación.
  - `MenuPrincipal` - Controlador principal.
  - `LanguageItem` - Modelo de idiomas.
- **Prefijos/Sufijos comunes:**
  - `Panel` - Para componentes de interfaz.
  - `Controller` - Para controladores FXML.
  - `Window` - Para gestores de ventanas.

8.2.3. Nombres de métodos

- **Convención:** camelCase.
- **Ejemplos:**
  - `calcularFirst()` - Calcula conjuntos PRIMERO.
  - `onBtnEditorAction()` - Manejador de eventos.
  - `cargarGramatica()` - Carga datos de gramática.
  - `actualizarTextos()` - Actualiza textos de interfaz.
- **Prefijos comunes:**
  - `calcular` - Para algoritmos de cálculo.
  - `cargar` - Para operaciones de carga.
  - `guardar` - Para operaciones de persistencia.
  - `on[Componente]Action` - Para manejadores de eventos.

8.2.4. Nombres de variables

- **Atributos de instancia:** camelCase con prefijo opcional.
  - `tabPane` - Panel de pestañas.
  - `gramaticaActual` - Gramática activa.
  - `bundle` - Recursos de internacionalización.
- **Variables locales:** camelCase descriptivo.
  - `conjuntoFirst` - Conjunto PRIMERO calculado.

  - `nuevaProduccion` - Nueva regla de producción.
- **Constantes:** UPPER_SNAKE_CASE.
  - `IDIOMA_POR_DEFECTO` - Idioma por defecto.
  - `MAXIMO_PRODUCCIONES` - Límite de producciones.

8.2.5. Nombres de archivos

- **Clases Java:** `NombreClase.java`.
- **Archivos FXML:** `NombreVentana.fxml`.
- **Archivos CSS:** `nombreEstilos.css`.
- **Propiedades i18n:** `messages_xx.properties`.
- **Recursos:** `nombre-descriptivo.ext`.

8.3. Estructura y organización del código

8.3.1. Estructura de clases

- **Orden de miembros:**
  1. Constantes estáticas.
  2. Atributos de instancia.
  3. Constructores.
  4. Métodos públicos.
  5. Métodos protegidos.
  6. Métodos privados.
  7. Métodos estáticos.
- **Agrupación lógica:** métodos relacionados se agrupan juntos.
- **Documentación:** JavaDoc completo para clases y métodos públicos.

8.3.2. Imports y dependencias

- **Orden de imports:**
  1. Imports de Java estándar.
  2. Imports de JavaFX.
  3. Imports de librerías externas (iText, etc.).
  4. Imports del proyecto actual.

- **Imports específicos:** evitar `import java.util.*;`.
- **Líneas en blanco:** separar grupos de imports relacionados.

8.4. Patrones de diseño implementados

8.4.1. Patrones creacionales

8.4.1.1. Singleton

El patrón *Singleton* asegura que una clase tenga únicamente una instancia y proporciona un punto de acceso global a ella. En SimAS 3.0, este patrón se utiliza para garantizar que la pantalla de bienvenida sea única durante toda la ejecución de la aplicación.

Implementación en SimAS 3.0: la clase `Bienvenida` implementa el patrón Singleton mediante:

- Un constructor privado que previene la instanciación externa.
- Un atributo estático privado que mantiene la única instancia.
- Un método estático público `getInstance()` que devuelve la instancia única.
- Inicialización lazy (diferida) para optimizar recursos.

Esta implementación permite que la aplicación tenga un único punto de entrada controlado, evitando múltiples instancias que podrían causar conflictos en la gestión de recursos de JavaFX y mantener el estado consistente de la aplicación.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/bienvenida/Bienvenida.java

Referencia: [2, pp. 127-134]

8.4.1.2. Factory Method

El patrón *Factory Method* define una interfaz para crear objetos, pero permite a las subclases decidir qué clase instanciar. En SimAS 3.0, este patrón facilita la creación dinámica de componentes de interfaz y objetos del modelo de datos.

Implementación en SimAS 3.0: el sistema utiliza Factory Methods en varios puntos:

- `TabManager.getOrCreateTab()` - Método fábrica que crea pestañas específicas según el tipo solicitado (editor, simulador, ayuda).
- Factorías para paneles de simulación que crean los diferentes pasos del asistente de simulación descendente.

- Creación de símbolos gramaticales (`Terminal`, `NoTerminal`) a través de métodos que encapsulan la lógica de instanciación.

Esta implementación permite que el código cliente no conozca las clases concretas que se están creando, proporcionando flexibilidad para añadir nuevos tipos de componentes sin modificar el código existente.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/TabManager.java

Referencia: [2, pp. 107-116]

8.4.2. Patrones estructurales

8.4.2.1. MVC (Modelo-Vista-Controlador)

El patrón *Modelo-Vista-Controlador* (MVC) separa la lógica de aplicación en tres componentes interconectados: el modelo (datos y lógica de negocio), la vista (presentación) y el controlador (manejo de entrada del usuario).

Implementación en SimAS 3.0: la arquitectura completa de la aplicación sigue el patrón MVC:

- **Modelo:** el paquete `gramatica` contiene todas las clases de dominio (`Gramatica`, `Simbolo`, `TablaPredictiva`) que representan los datos y la lógica de negocio del analizador sintáctico.
- **Vista:** el paquete `vistas` contiene los archivos FXML que definen la estructura visual de las interfaces, junto con los estilos CSS que controlan la apariencia.
- **Controlador:** las clases en paquetes como `bienvenida`, `editor`, `simulador` y `utils` actúan como controladores, manejando los eventos del usuario y coordinando entre modelo y vista.

Esta separación permite que los cambios en la interfaz no afecten la lógica de negocio, facilitando el mantenimiento y la evolución del código. Por ejemplo, se pueden cambiar los estilos visuales sin modificar la lógica del analizador sintáctico.

Referencia: [4]

8.4.2.2. Facade

El patrón *Facade* proporciona una interfaz unificada para un conjunto de interfaces en un subsistema, simplificando su uso y ocultando su complejidad.

Implementación en SimAS 3.0: la clase `MenuPrincipal` actúa como fachada del sistema, proporcionando una interfaz simple para acceder a todas las funcionalidades de la aplicación:

- Centraliza la navegación entre los diferentes módulos (editor, simulador, ayuda).
- Proporciona métodos de alto nivel que encapsulan operaciones complejas de inicialización.
- Oculta la complejidad de la gestión de pestañas y componentes internos.
- Ofrece una API consistente para que otros componentes interactúen con el sistema principal.

Por ejemplo, en lugar de que los componentes tengan que conocer cómo crear pestañas específicas o gestionar el estado de la aplicación, simplemente llaman métodos del `MenuPrincipal` que maneja toda la complejidad interna.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/bienvenida/MenuPrincipal.java

Referencia: [2, pp. 185-193]

8.4.2.3. Composite

El patrón *Composite* permite tratar objetos individuales y composiciones de objetos de manera uniforme, creando una estructura de árbol donde tanto las hojas como los nodos compuestos implementan la misma interfaz.

Implementación en SimAS 3.0: la jerarquía de símbolos gramaticales utiliza el patrón Composite para representar la estructura de las gramáticas:

- `Simbolo` es la clase base abstracta (Component) que define la interfaz común.
- `Terminal` y `NoTerminal` son las implementaciones concretas (Leaf y Composite respectivamente).
- Los símbolos no terminales pueden contener otros símbolos (producciones), creando una estructura jerárquica.
- El código cliente puede tratar todos los símbolos de manera uniforme sin conocer su tipo específico.

Esta implementación permite que algoritmos como la construcción de tablas predictivas o la validación de gramáticas puedan operar recursivamente sobre la estructura de símbolos sin necesidad de conocer si están trabajando con terminales o no terminales.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/gramatica/Simbolo.java

Referencia: [2, pp. 163-173]

8.4.3. Patrones de comportamiento

8.4.3.1. Observer

El patrón *Observer* define una dependencia de uno-a-muchos entre objetos, de manera que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.

Implementación en SimAS 3.0: el sistema de internacionalización utiliza el patrón Observer para mantener sincronizados todos los textos de la interfaz cuando cambia el idioma:

- La interfaz `ActualizableTextos` define el contrato que deben implementar todos los observadores (componentes de UI).
- Cada componente visual (botones, etiquetas, menús) implementa esta interfaz y registra sus textos observables.
- Cuando el usuario cambia el idioma, el sistema notifica a todos los observadores registrados.
- Cada componente actualiza automáticamente sus textos desde el archivo de propiedades correspondiente.

Esta implementación permite que el cambio de idioma sea instantáneo en toda la aplicación sin necesidad de reiniciar, y asegura que todos los componentes estén siempre sincronizados con el idioma seleccionado por el usuario.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/utils/ActualizableTextos.java

Referencia: [2, pp. 293-303]

8.4.3.2. Strategy

El patrón *Strategy* define una familia de algoritmos, encapsula cada uno y los hace intercambiables, permitiendo que el algoritmo varíe independientemente de los clientes que lo utilizan.

Implementación en SimAS 3.0: el sistema utiliza el patrón Strategy para manejar diferentes algoritmos de procesamiento de gramáticas:

- **Estrategias de análisis sintáctico:** diferentes algoritmos para analizar gramáticas (LL(1), otros métodos predictivos).
- **Estrategias de manejo de errores:** diferentes enfoques para tratar errores durante el análisis sintáctico.
- **Algoritmos de transformación:** estrategias para convertir gramáticas a formas normales o optimizadas.

- **Estrategias de visualización:** diferentes formas de mostrar los resultados de simulación.

Esta implementación permite cambiar dinámicamente el comportamiento del analizador sin modificar el código cliente, facilitando la extensión del sistema con nuevos algoritmos y la comparación de diferentes enfoques de resolución.

Referencia: [2, pp. 315-323]

8.4.3.3. Command

El patrón *Command* encapsula una solicitud como un objeto, permitiendo parametrizar clientes con diferentes solicitudes, encolar solicitudes, y soportar operaciones de deshacer/rehacer.

Implementación en SimAS 3.0: las operaciones de simulación utilizan el patrón Command para manejar acciones complejas de manera estructurada:

- **Comandos de navegación:** avanzar/retroceder en los pasos de la simulación descendente.
- **Operaciones de edición:** funcionalidad de deshacer/rehacer para modificaciones en gramáticas.
- **Acciones del menú contextual:** comandos que pueden ejecutarse desde diferentes contextos (botones, menús, atajos de teclado).
- **Comandos de simulación:** ejecutar análisis sintáctico, mostrar resultados, generar reportes.

Cada comando implementa una interfaz común que permite ejecutarlo, deshacerlo y verificar si puede ejecutarse, proporcionando consistencia en el manejo de todas las operaciones del sistema.

Referencia: [2, pp. 233-242]

8.4.3.4. Template Method

El patrón *Template Method* define el esqueleto de un algoritmo en una operación, delegando algunos pasos a las subclases, permitiendo que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

Implementación en SimAS 3.0: los asistentes de simulación utilizan el patrón Template Method para mantener una estructura consistente en todos los pasos:

- `PanelNuevaSimDescPaso` define el método plantilla con la estructura común de todos los pasos.

- Cada paso específico (`Paso1`, `Paso2`, etc.) implementa los métodos específicos para su funcionalidad.
- **Estructura común:** inicialización, validación de entrada, procesamiento, actualización de UI, navegación.
- **Especializaciones:** contenido específico de cada paso (configuración de gramática, definición de entrada, etc.)

Esta implementación asegura que todos los pasos del asistente sigan el mismo flujo de trabajo, manteniendo consistencia en la experiencia del usuario mientras permiten flexibilidad en el contenido específico de cada paso.

Referencia: [2, pp. 325-330]

8.4.3.5. Mediator

El patrón *Mediator* define cómo un conjunto de objetos interactúan entre sí, promoviendo el bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, centralizando en su lugar la interacción.

Implementación en SimAS 3.0: varias clases actúan como mediadores para coordinar la comunicación entre componentes complejos:

- `Editor` actúa como mediador entre todos los paneles del asistente de creación de gramáticas, coordinando su activación/desactivación y el flujo de datos entre ellos.
- `SimulacionFinal` gestiona la comunicación entre todos los componentes de la simulación (paneles, controles, visualización de resultados).
- `TabManager` coordina la jerarquía de pestañas, manejando la creación, cierre y navegación entre diferentes vistas del sistema.
- **Beneficio:** los componentes individuales no necesitan conocer cómo interactuar directamente con otros componentes.

Esta implementación reduce significativamente el acoplamiento entre componentes, facilitando el mantenimiento y la evolución del código, ya que los cambios en la lógica de coordinación se centralizan en los mediadores.

Ubicación: https://github.com/Llamatekee/SimAS-3.0/blob/main/src/editor/Editor.java

Referencia: [2, pp. 273-282]

8.5. Mejores prácticas implementadas

8.5.1. Gestión de memoria

- **JavaFX Properties:** para binding automático de datos.

  - `StringProperty` para textos observables.
  - `IntegerProperty` para contadores.
  - `BooleanProperty` para estados.
- **Collections observables:** para actualización automática de UI.
  - `FXCollections.observableArrayList()`.
  - `FXCollections.observableMap()`.
- **Lazy loading:** componentes cargados bajo demanda.

8.5.2. Manejo de errores

- **Validación temprana:** verificación de entrada en UI.
- **Mensajes informativos:** diálogos contextuales para usuarios.
- **Logging estructurado:** información de debugging completa.
- **Recuperación graceful:** manejo de errores sin crash.

8.5.3. Internacionalización

- **ResourceBundle:** sistema estándar de Java.
- **UTF-8 completo:** soporte para caracteres internacionales.
- **Actualización dinámica:** cambio de idioma sin reinicio.
- **Separación clara:** textos separados del código.

8.5.4. Documentación

- **JavaDoc completo:** para todas las clases y métodos públicos.
- **Comentarios explicativos:** algoritmos complejos documentados.
- **Referencias cruzadas:** enlaces entre componentes relacionados.
- **Ejemplos de uso:** casos de uso documentados.

8.6. Principios de diseño aplicados

8.6.1. SOLID Principles

8.6.1.1. S - Single Responsibility Principle

Cada clase tiene una única responsabilidad bien definida:

- `Gramatica` - Gestiona datos y algoritmos gramaticales.
- `SimulacionFinal` - Maneja simulación paso a paso.
- `TabManager` - Coordina pestañas jerárquicas.

8.6.1.2. O - Open/Closed Principle

Las clases están abiertas a extensión pero cerradas a modificación:

- Interfaces como `ActualizableTextos`.
- Jerarquía extensible de símbolos gramaticales.
- Sistema de plugins para funciones de error.

8.6.1.3. L - Liskov Substitution Principle

Las subclases pueden reemplazar a sus clases base:

- `Terminal` y `NoTerminal` extienden `Simbolo`.
- Implementaciones de `ActualizableTextos`.

8.6.1.4. I - Interface Segregation Principle

Interfaces específicas en lugar de generales:

- `ActualizableTextos` - Solo para internacionalización.
- Interfaces específicas por funcionalidad.

8.6.1.5. D - Dependency Inversion Principle

Dependencia de abstracciones, no de concreciones:

- Inyección de dependencias en controladores.
- Interfaces para servicios transversales.

8.6.2. DRY (Don't Repeat Yourself)

- **Componentes reutilizables:** paneles base para asistentes.
- **Métodos utilitarios:** funciones comunes extraídas.
- **Constantes compartidas:** valores comunes centralizados.

8.6.3. KISS (Keep It Simple, Stupid)

- **Algoritmos claros:** implementaciones directas y comprensibles.
- **Interfaces intuitivas:** diseño de usuario simple.
- **Estructuras lógicas:** organización clara del código.

8.7. Herramientas de desarrollo

8.7.1. Control de versiones

- **Git:** sistema de control de versiones distribuido.
- **GitHub:** plataforma de colaboración y hosting.
- **Conventional commits:** formato estandarizado para mensajes.

8.7.2. IDE y herramientas

- **IntelliJ IDEA:** IDE principal para desarrollo.
- **JavaFX Scene Builder:** diseño de interfaces FXML.
- **Graphviz:** generación de diagramas (árboles sintácticos).

8.7.3. Construcción y despliegue

- **jpackage:** creación de ejecutables nativos.
- **JavaFX SDK:** framework de interfaces de usuario.
- **iText PDF:** generación de informes PDF.

8.8. Métricas de calidad del código

8.8.1. Complejidad ciclomática

- **Métodos simples:** $<$10 puntos de complejidad.
- **Métodos complejos:** 10-20 puntos (algoritmos de análisis).
- **Límites aceptables:** Máximo 25 puntos por método.

8.8.2. Cobertura de código

- **Enfoque:** casos de uso principales y algoritmos críticos.
- **Herramientas:** JaCoCo para medición de cobertura.

8.9. Guías de contribución

8.9.1. Proceso de desarrollo

1. **Fork** del repositorio principal.
2. **Clone** del fork local.
3. **Branch** para nueva funcionalidad.
4. **Commits** frecuentes con mensajes descriptivos.
5. **Pull request** para revisión.
6. **Code review** y aprobación.
7. **Merge** a rama principal.

8.9.2. Estandares de commit

- **Formato:** `tipo: descripción`.
- **Tipos comunes:**
  - `feat:` nueva funcionalidad.
  - `fix:` corrección de errores.
  - `docs:` cambios en documentación.
  - `style:` cambios de formato/código.
  - `refactor:` refactorización de código.

8.9.3. Code reviews

- **Revisión obligatoria:** Todos los cambios requieren aprobación.
- **Criterios de calidad:**
  - Funcionalidad correcta.
  - Código legible y documentado.
  - Pruebas incluidas.
  - Convenciones de código respetadas.

8.10. Conclusión

Las convenciones y mejores prácticas implementadas en SimAS 3.0 garantizan:

- **Mantenibilidad:** código fácil de entender y modificar.
- **Escalabilidad:** arquitectura que soporta crecimiento.
- **Calidad:** implementación de patrones probados.
- **Consistencia:** estándares uniformes en todo el proyecto.

Estas directrices no solo mejoran la calidad actual del código, sino que también facilitan la incorporación de nuevos desarrolladores y la evolución futura del sistema.

Bibliografía

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd. Addison-Wesley, 2007. ISBN: 978-0321486813.

[2] E. Gamma, R. Helm, R. Johnson y J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st. Conocido como "Gang of Four.º libro de patrones GoF. Addison-Wesley Professional, 1995. ISBN: 978-0201633610.

[3] J. E. Hopcroft, R. Motwani y J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd. Addison-Wesley, 2007. ISBN: 978-0321455369.

[4] G. E. Krasner y S. T. Pope. «A description of the model-view-controller user interface paradigm in the Smalltalk-80 system». En: *Journal of Object-Oriented Programming* 1.3 (1988), págs. 26-49.

[5] OpenJFX Project. *JavaFX - Open Source Java Client Platform*. [En Línea. Última consulta: 20-12-2024]. URL: https://openjfx.io/.

[6] Oracle Corporation. *Java Platform, Standard Edition*. [En Línea. Última consulta: 20-12-2024]. URL: https://www.oracle.com/java/.

[7] Oracle Corporation. *jpackage - Java Platform, Standard Edition Tools Reference*. [En Línea. Última consulta: 20-12-2024]. URL: https://docs.oracle.com/en/java/javase/17/docs/specs/man/jpackage.html.