Documento de Arquitectura de Software

Profesora: Maria Estrada.

Curso: Proyecto de Ingeniería de Software.

11.01.2017

Jose Fernando Molina Chacón

Yordan Jiménez Hernández

Contenidos

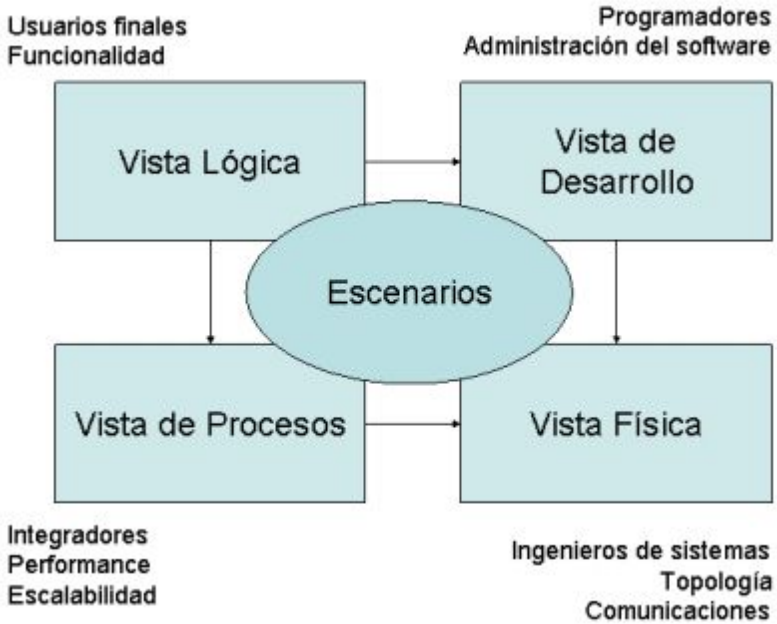
1. Introducción

El siguiente documento provee una visión de conjunto y explica de manera completa la arquitectura del software Simplex Educativo, desarrollado para el profesor José Helo como una herramienta educativa que pueda utilizar como apoyo para el desarrollo de la clase de Investigación de Operaciones. Explica también la manera de operación del software, las interacciones entre los diferentes objetos y componentes así como la arquitectura subyacente de la solución. El presente escrito contiene una descripción de alto nivel de los casos de uso que deben ser cubiertos, así como como las decisiones arquitecturales que se tomaron para cumplir dichos objetivos.

1.1. Propósito

El propósito de este documento es proveer al lector con la información arquitectural del software Simplex Educativo de manera que pueda comprender la manera en que el producto fue construida y las diferentes decisiones que se tomaron para la implementación de la solución. Por medio de 4 diferentes vistas del mismo problema se espera que el lector logre comprender la manera en que los casos de uso fueron materializados en artefactos de software para cumplir la funcionalidad buscada, así como explicar los aspectos relevantes de la arquitectura que pueden no ser intuitivos para una persona no familiarizada con el software.

Para la mayor claridad posible para el siguiente documento se utilizó el modelo 4+1 desarrollado por Philippe Kruchten de descripción arquitectural.



1.2. Alcance

Este documento se basa en el modelo 4+1 presentado anteriormente, y describe cada una de las vistas de este modelo aplicadas al proyecto Simplex Educativo. Describe los aspectos de la herramienta que se consideran arquitecturalmente significativos, esto es, los elementos y comportamientos que son fundamentales para comprender el proceso de construcción del producto final y su entendimiento como un todo. La descripción se realizará solamente tomando en cuenta los casos de uso escogidos para la iteración número 1 del actual proyecto.

1.3. Definiciones y acrónimos

- **ERS:** Especificación de Requerimientos de Software.
- **MVC:** Model-View-Controller. Patrón de diseño arquitectural.
- **Simplex:** Algoritmo creado por George Dantzig para resolver problemas de programación lineal.
- **Programación Lineal:** Modelo matemático para buscar la optimización de una función objetivo que cumpla con N restricciones al mismo tiempo.
- **Java:** Lenguaje de programación orientado a objetos.
- **JDK**: Java Development Kit. Herramientas de desarrollo del lenguaje Java.
- **UML:** Unified Modeling Language. Lenguaje gráfico descriptivo para uso en procesos de software.
- **Problema Infactible:** Problema de programación lineal que debido a sus restricciones hace que la función objetivo no se pueda cumplir bajo ninguna circunstancia.
- **Problema No Acotado:** Problema de programación lineal que debido a sus restricciones hace que la función objetivo pueda ser maximizada o minimizada de manera infinita (no existe límite para detenerse).
- **BVS:** Basic Variables. Variables básicas, es decir, variables que poseen un valor definido en cierto momento.
- **RHS:** Right Hand Side. Lado derecho de una igualdad o desigualdad. En representación matricial simplex, la última columna de derecha a izquierda.
- **Radios:** Resultado de dividir el RHS entre una columna N de una matriz simplex. Tienen la particularidad de que si la división es entre 0 o el resultado es un número negativo, el resultado de los radios será infinito.
- **Pivotear:** Tomar una entrada diferente de cero de una matriz y realizar operaciones fila de manera que la columna de esa entrada sea básica, es decir, solamente posea 0ñs y solamente un 1 en la posición de pivote escogida.

1.4. Referencias

- Documento de Visión de Simplex Educativo.
- Especificación de Requerimientos de Software de Simplex Educativo.

1.5. Resumen

El documento está dividido en las siguientes secciones:

Sección 2: Describe cada una de las vistas propuestas para la explicación de la arquitectura.
Sección 3: Describe las restricciones arquitecturales del sistema.
Sección 4: Describe los casos de uso.
Sección 5: Describe los paquetes arquitecturales y las clases utilizadas
Sección 6: Describe los procesos que ejecutan los casos de uso.
Sección 7: Describe los componentes que interactúan para cumplir los casos de uso.
Sección 8: Describe el despliegue fisico del sistema.
Sección 9: Describe la capa de persistencia de datos y como fueron cumplidos sus requerimientos.

2. Representación de la arquitectura

Las siguientes vistas representan la arquitectura del sistema para la primera iteración:

1. **Vista de casos de uso:**
    a. **Audiencia:** Todos los stakeholders, incluyendo al usuario final.
    b. **Descripción:** Describe el conjunto de escenarios y/o casos de uso que representen alguna funcionalidad principal o central del sistema, así como los usuarios del mismo. Esta vista presenta las necesidades del usuario y es utilizada en vistas posteriores como la lógica o de procesos. Los casos de uso describen de manera muy general uno de los fines que tiene el usuario con el sistema, es decir, una acción que el usuario desea completar con el software.
    c. **Artefactos relacionados:** Diagramas de Casos de Uso.
2. **Vista de Lógica:**
    a. **Audiencia:** Diseñadores, equipo de desarrollo.
    b. **Descripción:** Esta vista describe la funcionalidad de la aplicación en términos de elementos estructurales, abstracciones clave y mecanismos, separación y distribución de responsabilidades. Se definirán los distintos niveles lógicos del sistema.
    c. **Artefactos relacionados:** Diagrama de Clase, Diagrama de Paquetes.
3. **Vista de Procesos:**
    a. **Audiencia:** Integradores, equipo de desarrollo.

   b. **Descripción:** Considera los requerimientos no funcionales como eficiencia, escalabilidad y afines. Muestra las abstracciones de la vista lógica y su operación entre ellos de manera temporal en un hilo de ejecución (la manera y el orden en que las abstracciones colaboran entre sí). Un proceso es un grupo de tareas que forman una unidad ejecutable, en este caso, un caso de uso. En resumen, la vista de procesos describe la manera en que las abstracciones de la vista lógica interactúan entre sí para cumplir los casos de uso.
   c. **Artefactos relacionados:** Diagramas de actividad
4. **Vista de Desarrollo:**
   a. **Audiencia:** Equipo de desarrollo.
   b. **Descripción:** Es la vista de la arquitectura del sistema que encierra los componentes para armar y lanzar un sistema. Se concentra en todos los componentes que posee un sistema y la manera que dichos componentes se relacionan entre sí por medio de interfaces para lograr un objetivo específico.
   c. **Artefactos Relacionados:** Diagrama de Componentes.
5. **Vista de Despliegue**
   a. Equipo de Desarrollo, técnicos en TI.
   b. Mientras que la vista lógica se encarga de describir los componentes lógicos del sistema, la vista de despliegue muestra los componentes físicos de hardware y su interacción .
   c. Artefactos relacionados: Diagrama de Despliegue.

3. Metas y restricciones arquitecturales.

3.1. Principios SOLID

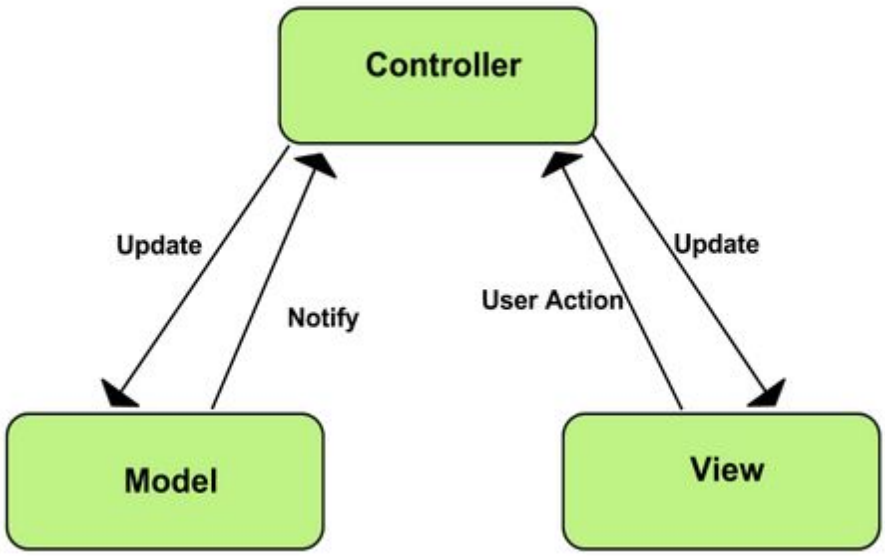
Para el diseño arquitectural se debe tomar en cuenta los principios SOLID para diseño de software. Dichos principios se describen a continuación:

- **Single Responsability Principle:** Toda entidad de software tendrá una y solamente una responsabilidad.
- **Open-Closed principle:** Las entidades deben estar cerradas para modificación pero abiertas para extensión.
- **Liskov Substitution Principle:** Los objetos de tipo T pueden ser sustituidos por objetos de tipo S siempre que S sea un subtipo de T. Esto implica que las interfaces públicas de subclases deben ser iguales a las de su superclase.
- **Interface Segregation Principle:** Los objetos no deben depender de métodos que no utilicen.
- **Dependency Inversion Principle:** Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben

depender de abstracciones. Asimismo, las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones. Esto implica que las clases concretas no deben tener referencias a objetos concretos, sino a sus abstracciones.

3.2. Implementación de MVC.

Para la implementación de la solución se debe utilizar el patrón arquitectural Model-View-Controller. En este patrón existen diferentes tipos de objetos que pertenecen a alguna de las categorías anteriores: en el modelo está la lógica de negocio, en la vista las interfaces gráficas para interacción con el usuario y el controlador es el encargado de recoger las acciones del usuario en la interfaz gráfica y modificar el modelo de manera acorde. La variación del mismo será la siguiente:



Este tipo de MVC provee una mayor independencia entre el modelo y la vista, pues el modelo no debe conocer nada de la vista, de ello se encarga el controlador, el cual la actualizará acordemente dependiendo de las acciones realizadas por el usuario.

3.3. Herramientas gratuitas.

El proyecto será realizado sin remuneración económica, por este motivo todas las herramientas que se utilicen para el desarrollo del mismo deben ser gratuitas para evitar incurrir en costos innecesarios por parte del equipo de desarrollo.

3.4. Exclusión de la eficiencia.

La eficiencia del algoritmo Simplex no es una restricción. El software será utilizado para la educación y no para la resolución de problemas

complicados de programación lineal, por lo cual las restricciones de eficiencia son despreciables.

3.5. Portabilidad

La solución realizada debe ser portable hacia dispositivos independientes del sistema operativo que posean. Es por ello que se decide utilizar el lenguaje de programación Java para desarrollar el proyecto. Se debe tomar en cuenta que los objetos que se deban utilizar pertenezcan a las librerías clásicas del JDK o a librerías desarrolladas en Java completamente y que dicha librerías no sean dependientes del sistema operativo.

4. Vista de casos de uso

4.1. Casos de Uso

4.1.1. Verificar la factibilidad de un problema de programación lineal ingresado.

Nombre	Verificar la factibilidad de un problema de programación lineal ingresado.
Actor	Usuario final
Sinopsis	El caso inicia cuando el usuario después de ingresar un problema de programación lineal, solicita al sistema su solución por medio del método simplex. Durante este último paso mencionado, se podrá encontrar la característica de que el sistema no posea una solución óptima factible y el usuario debe que ser notificado.

4.1.2. Obtener de manera inmediata una solución óptima de un problema de programación lineal.

Nombre	Obtener de manera inmediata una solución óptima de un problema de programación lineal.
Actor	Usuario Final
Sinopsis	Luego de que el usuario ingresa un problema de programación lineal en el campo respectivo, se podrá obtener la primera solución óptima para el problema ingresado, donde se desplegará la última iteración generada junto a la información de la solución y un resumen que liste

	los pasos realizados.

4.1.3. Listar los pasos intermedios para encontrar la solución a un problema de programación lineal.

Nombre	Listar los pasos intermedios para encontrar la solución a un problema de programación lineal.
Actor	Usuario Final
Sinopsis	El usuario podrá ingresar un problema de programación lineal, y obtener su solución navegando entre todas las iteraciones generadas por el método Simplex, donde cuando se encuentre la primera solución óptima se notificará al usuario o, si se encontró la característica de que el problema no era factible o no se encuentra acotado, se le notifique por medio de un mensaje.

4.1.4. Verificar si un problema de programación lineal ingresado se encuentra acotado.

Nombre	Verificar si un problema de programación lineal ingresado se encuentra acotado.
Actor	Usuario Final
Sinopsis	El caso inicia cuando el usuario después de ingresar un problema de programación lineal, solicita al sistema su solución por medio del método simplex. Durante este último paso mencionado, se podrá encontrar la característica de que el sistema no posea una solución óptima acotada dentro de un rango y el usuario tenga que ser notificado mediante un mensaje.

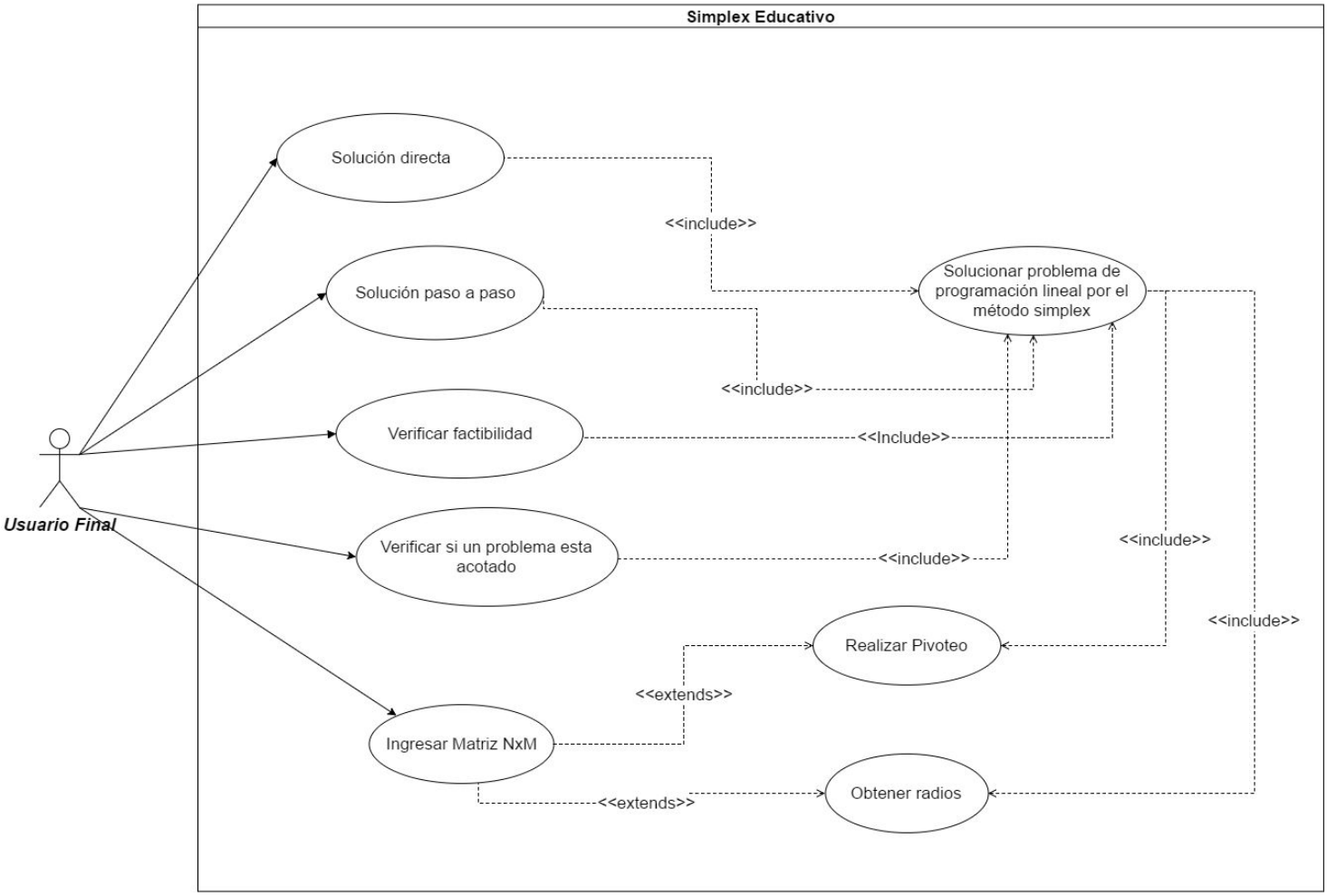
4.1.5. Ingresar una matriz de N x M manualmente donde se podrá realizar una operación de fila.

Nombre	Ingresar una matriz de N x M manualmente donde se podrá realizar una operación de fila o obtener los radios entre una columna y el "*lado derecho*" de la matriz.
Actor	Usuario Final
Sinopsis	El usuario podrá ingresar un problema de programación lineal, donde el transcurso de generar la solución de este mismo, tendrá la capacidad de seleccionar donde realizar pivoteos o obtener el radio tras seleccionar una columna para dicho proceso. Durante este proceso el usuario podrá observar si el problema se vuelve no factible o no se encuentra acotado.

4.1.6. Obtener los radios entre una columna y el "*lado derecho*" de la matriz

Nombre	Obtener los radios entre la columna y el lado derecho de la matriz.
Actor	Usuario Final
Sinopsis	El usuario podrá ingresar un problema de programación lineal, donde el transcurso de generar la solución de este mismo, tendrá la capacidad de seleccionar una columna para calcular los radios de dicha columna con el RHS de la matriz.

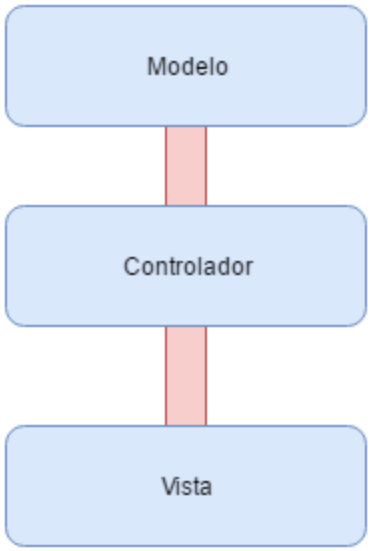
4.2. Diagrama de Casos de Uso



5. Vista logica

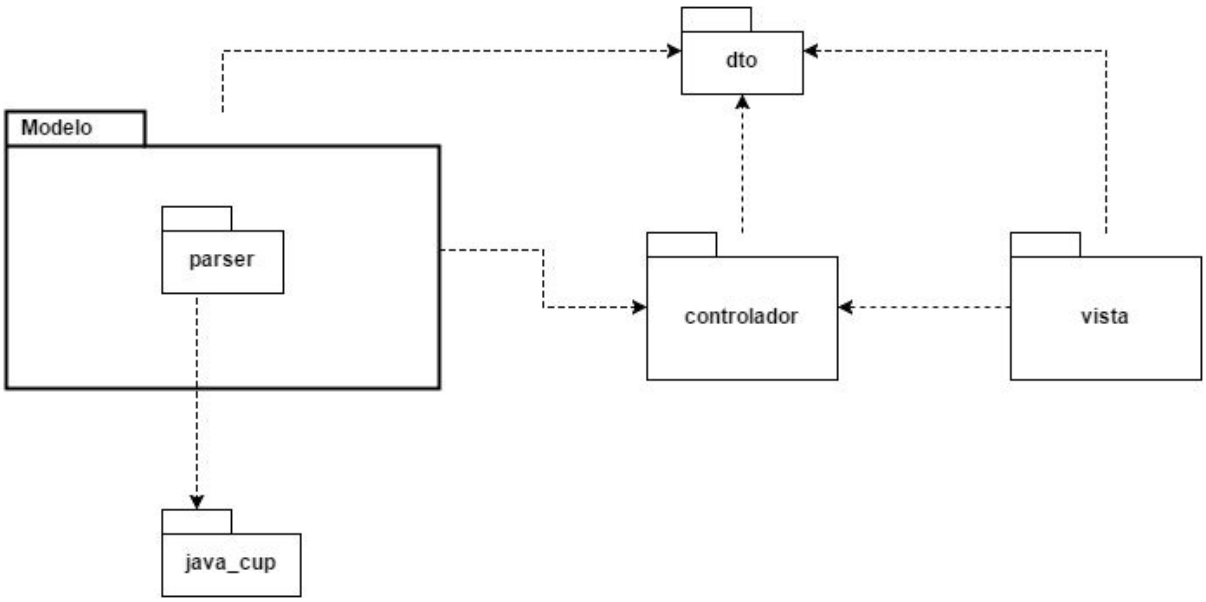
5.1. Vista general

Debido a que el programa por desarrollar no es significativamente complicado en términos de responsabilidades, cantidad de clases ni extensión se decidió realizar la división de las capas de acuerdo a la función que cumplen respecto al patrón MVC. Esta estrategia ha sido escogida debido a que aisla las responsabilidades de cada clase dependiendo solamente de su función dentro del sistema respecto al patrón mencionado. La división lógica de las capas del sistema Simplex Educativo es la siguiente. El diagrama de clases completo puede ser encontrado en el repositorio.



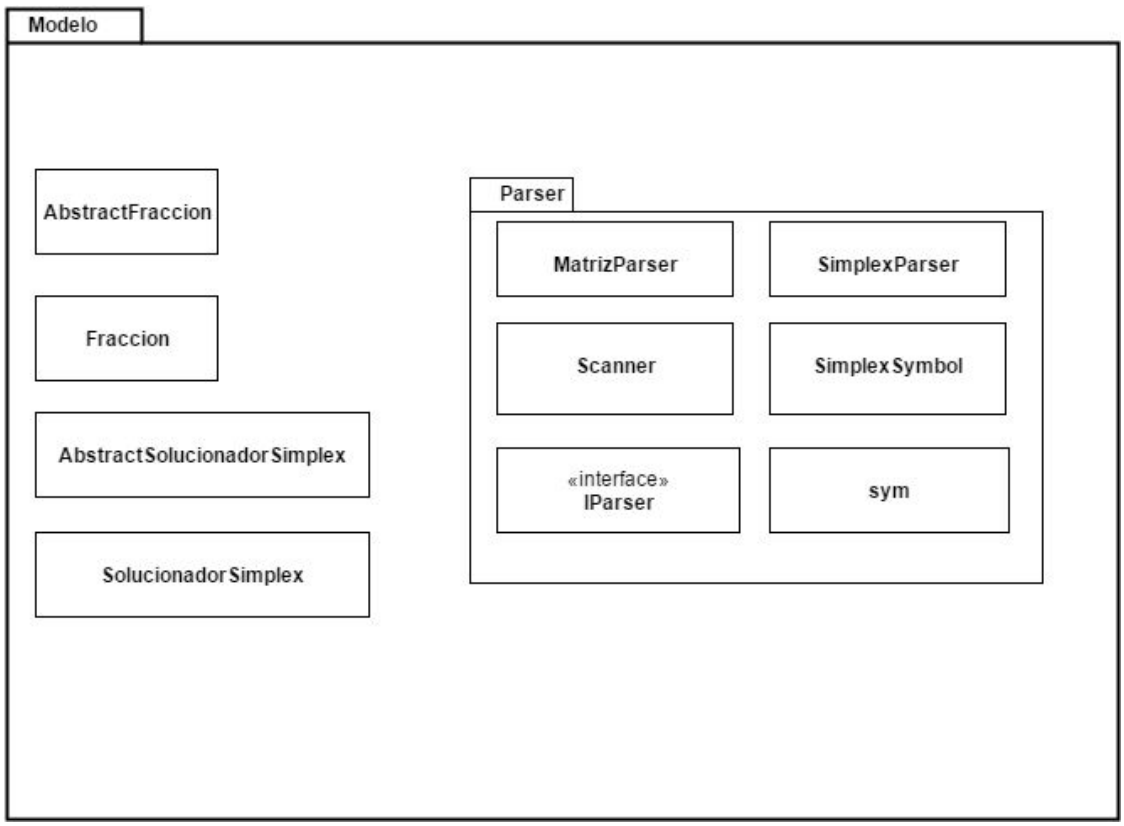
- **Modelo:** Es la capa encargada de representar y manejar el comportamiento de los datos solamente. Ella responde a estímulos por parte de la capa controlador únicamente. Es en esta capa donde se encuentran todas las clases necesarias para la ejecución de los casos de uso a nivel lógico meramente, sin control por parte del usuario ni interfaz gráfica de por medio.
- **Controlador:** Capa encargada de controlar la interacción entre la vista y el modelo. Es esta capa la cual recibe los estímulos por parte del usuario y comunica al modelo la acción pedida, para luego tomar la respuesta del modelo y presentarselas a la vista para su despliegue.
- **Vista:** Capa encargada de la representación gráfica del modelo. Utiliza al controlador para hacer pedidos al modelo, y el controlador se encarga de actualizar la vista acordemente.

5.2. Paquetes Arquitecturales



5.2.1. Modelo

Contiene todas las clases que representan el modelo de la solución. Es el paquete encargado del funcionamiento lógico de la aplicación, siendo él el encargado de proveer las funcionalidades necesarias para resolver los casos de uso descritos anteriormente.



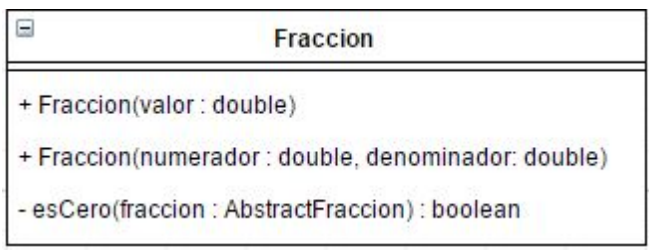
Clases e Interfaces

- **AbstractFraccion:**

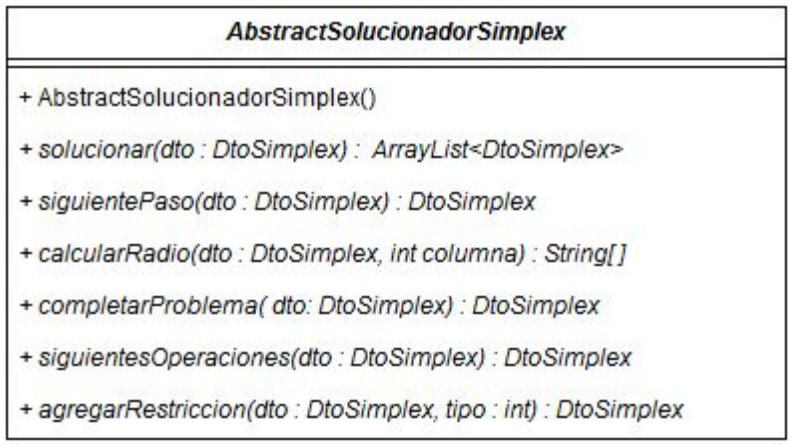
- **Descripción:** Definición de la clase abstracta que simboliza una fracción matemática. Posee métodos para manejar matemáticamente una representación de fracciones sin perder precisión decimal al tratar a los números reales como partes enteras para de esta manera realizar operaciones sobre fracciones y no en punto flotante.

- **UML:**

- **Fracción:**
  - **Descripcion:** Implementación concreta de AbstractFraccion.
  - **UML:**



- **AbstractSolucionadorSimplex:**
  - **Descripción:** Implementación abstracta de los métodos básicos necesarios para resolver un problema de programación lineal cumpliendo con los requerimientos del cliente.
  - **UML:**



- **SolucionadorSimplex:**
  - **Descripción:** Posee la implementación de los métodos de AbstractSolucionadorSimplex. Clase principal del programa, posee muchos métodos privados necesarios para la resolución del simplex por lo cual la clase es extensa.

- **UML:**

SolucionadorSimplex

+ SolucionadorSimplex()

+ calcularRadio(dto : DtoSimplex, int columna) : String[]

+ siguientePaso(dto : DtoSimplex) : DtoSimplex

+ solucionar(dto : DtoSimplex) : ArrayList<DtoSimplex>

-_SiguientePaso(dto : DtoSimplex) : DtoSimplex

+ completarProblema(dto: DtoSimplex) : DtoSimplex

+ siguientesOperaciones(dto : DtoSimplex) : DtoSimplex

- obtenerIndiceDelValorMenor(valores: AbstractFraccion[ ], indiceInicio : int, acotoFinal : int) : int

- obtenerRadios (fracciones : AbstractFraccion[ ] [ ], columna : int) : AbstractFraccion [ ]

- generarRadio (ladoDerecho : AbstractFraccion, columna : AbstractFraccion) : AbstractFraccion

- pivotear(fracciones : AbstractFraccion[ ] [ ], fila1 : int, fila2 : int, valorOperador : AbstractFraccion) : AbstractFraccion[ ]

- verificarFactibilidad(valores : AbstractFraccion[ ]) : boolean

- esAcotado(ladoDerecho : AbstractFraccion[ ]) : boolean

- generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fila2 : int, fraccional : boolean, dosFases : boolean):String

- generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fraccional : boolean):String

- generarUno(fila : AbstractFraccion[ ], indiceElemento : int):AbstractFraccion[ ]

- realizarOperaciones(matriz : AbstraccionFraccion[ ] [ ], indiceFilaEntrante : int, indiceColumnaEntrante : int) : AbstraccionFraccion[ ] [ ]

- validarSimplexTerminado(funcionObjetivo : AbstraccionFraccion[ ]) : boolean

- negarCoeficientes(coeficientes : AbstraccionFraccion[ ]) : AbstraccionFraccion[ ]

- agregarUnoMatriz(matriz : AbstraccionFraccion[ ] [ ], int fila, int columna, boolean positivo):AbstraccionFraccion[ ] [ ]

agregarColumnas(matriz : AbstraccionFraccion[ ] [ ], int cantidad) : AbstraccionFraccion[ ] [ ]

- agregarUnos(matriz : AbstraccionFraccion[ ] [ ], posiciones : ArrayList<Integer>, positivo : boolean, inicio : int) : AbstraccionFraccion[ ] [ ]

convertirDosFases(matriz : AbstraccionFraccion[ ] [ ], int inicoArtificiales):AbstraccionFraccion[ ] [ ]

- crearFuncionW(tamano : int, inicioArtificiales : int) : AbstraccionFraccion[ ]

- generarCoeficiente(matriz : AbstractFraccion[][], indiceFila : int, indiceColumna : int) : AbstractFraccion

- siguientePivoteo(dto : DtoSimplex): Point

- agregarNombreVariables(nombres : String[], cantidad : int, nuevoNombre : String) : String[]

- agregarNombresFila(nombres : String[], Indicesfila : ArrayList<Integer>, nombreFila : String, indiceInicio : int) : String[]

- agregarNombreW(nombres : String []) : String[]

- crearNombreFila(tamano : int, nombreCabeza :String) : String[]

- siguientePasoSimplex(dto : DtoSimplex) : DtoSimplex

- siguientePasoDosFases(dto : DtoSimplex) : DtoSimplex

- eliminarArtificiales(dto : DtoSimplex) : DtoSimplex

buscarIndice(nombres : String[], nombre : String) : int

- eliminarFilaW(dto : DtoSimplex) : DtoSimplex

- eliminarColumnasArtificiales(dto : DtoSimplex) : DtoSimplex

- eliminarNombreW(nombresFila : String[]) : String []

- eliminarNombresColumnas(nombresColumnas : String[], indiceInicioArtificiales : int) : String[]

siguientesOperacionesInicioDosfases(indice :int) : String[]

obtenerSolucion(dto : DtoSimplex) : String

agregarFila(matriz : AbstractFraccion[][]) : AbstractFraccion[][]

- obtenerIndiceDeSiguienteVariableSaliente(valores, AbstractFraccion[], indiceInicio : int, acotoFinal : int, dtoProblema : DtoSimplex) : int

- agregarMenorIgual(dto : dtoSimplex) : DtoSimplex

- agregarMayorIgual(dto : dtoSimplex) : DtoSimplex

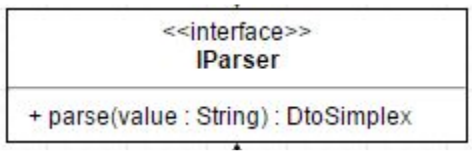
- agregarIgual(dto : dtoSimplex) : DtoSimplex

5.2.1.1. Parser

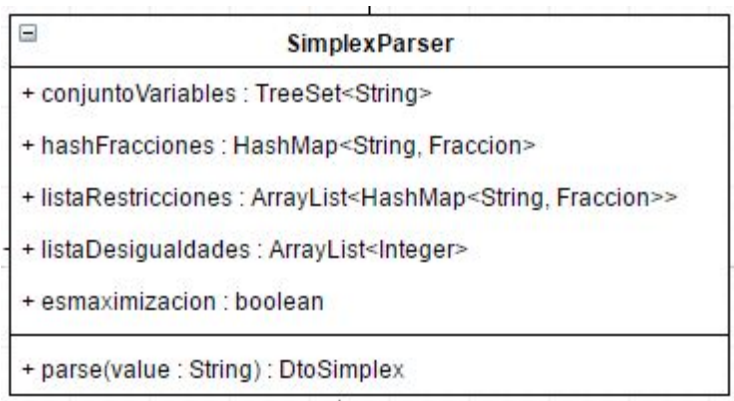
Posee las clases encargadas de interpretar una cadena de texto y convertirlo a un objeto que pueda ser utilizado por el programa, en este caso un DtoSimplex que contendrá toda la información necesaria para iniciar el proceso Simplex por parte de la clase SolucionadorSimplex. Este paquete posee 3 clases las cuales son generadas por herramientas externas: Scanner (JFlex), SimplexParser (CUP) y sym (CUP).

Clases e Interfaces

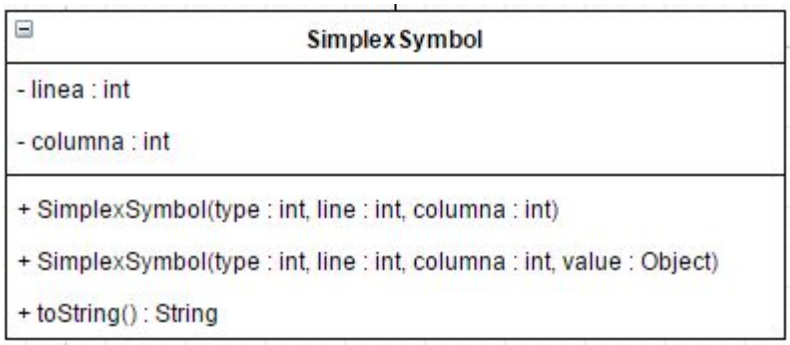
- **IParser:**
  - **Descripción:** Define el método parse(String) el cual se encarga de validar el correcto formato del string de entrada y retornar en un objeto DtoSimplex los valores necesarios del algoritmo para iniciar el análisis ya sea de Simplex o de pivoteos en una matriz.
  - **UML:**



- **SimplexParser:**
  - **Descripción:** Clase autogenerada por CUP. Se encarga de validar e interpretar una cadena de texto que contenga un problema de programación lineal con el formato válido aceptado por el programa.
  - **UML:** El siguiente diagrama UML solamente posee los métodos agregados por el equipo de desarrollo a la clase autogenerada, no incluye los métodos creados por el generador del código.



- **Scanner:**
  - **Descripción:** Clase autogenerada por JFLex. Se encarga de dividir el texto de entrada en tokens que puedan ser interpretados por el parser. Por ejemplo, la cadena de texto “3 x1 + ½ x2” identifica los tokens “3”, “x1”, “+”, “½ ” y “x2”. Por su parte, el Parser es el encargado de asegurarse que dichos tokens vengan en el orden correcto e interpretar el significado de la cadena de texto.
  - **UML:** No se incluye debido a que es una clase autogenerada.
- **SimplexSymbol:**
  - **Descripción:** Clase que representa un token de la cadena de texto del problema de programación lineal. Utilizado por Scanner y Parser para realizar sus operaciones. Subclase de java_cup.runtime.Symbol
  - **UML:**



- **sym:**
  - **Descripción:** Posee valores enteros constantes que representan los posibles símbolos reconocidos por el scanner. Cada tipo de token tiene asignado un valor entero que está definido en esa clase.
  - **UML:** Clase autogenerada. Solamente posee valores constantes enteros, no es necesario una representación en el diagrama UML.

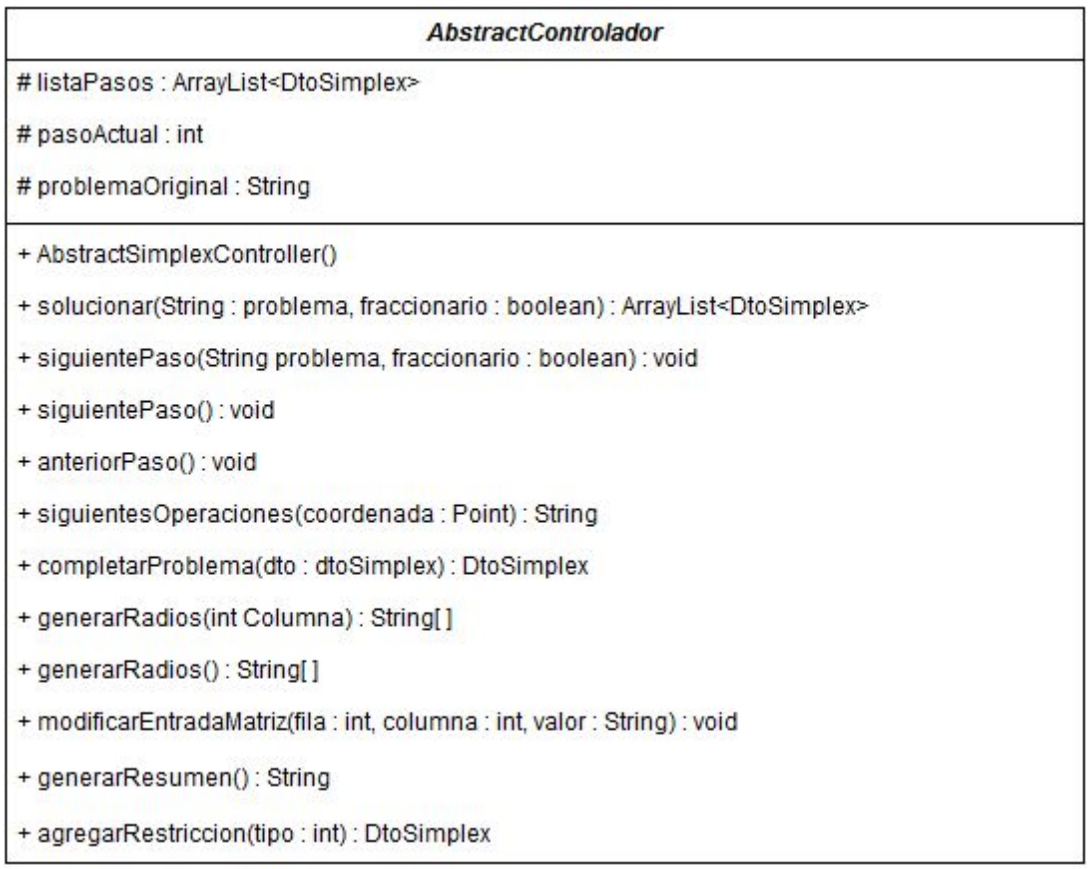
5.2.2. Controlador

En este paquete se encuentran dos controladores para los diferentes tipos de casos de uso que debieron ser realizados. También incluye la interfaz que debe implementar toda vista que quiera utilizar el modelo de Simplex Educativo para su reutilización. Esta interfaz se

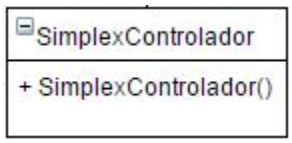
decidió ponerla en este paquete para eliminar la dependencia circular entre el paquete vista y el paquete controlador.

Clases e Interfaces

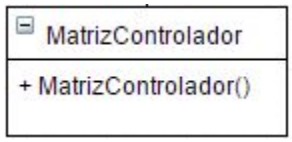
- **AbstractControlador:**
  - **Descripción:** Definición de la clase abstracta que simboliza un controlador para cumplir con los casos de uso junto a una interfaz gráfica. Posee métodos para manejar la interacción entre el modelo y la vista, por lo que contiene referencias a objetos de tipo AbstractSolucionadorSimplex y IVista. Es el encargado de tomar las peticiones de IVista, realizar la acción en el modelo y actualizar la vista con el nuevo estado del modelo.
  - **UML:**



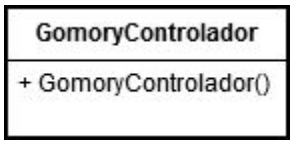
- **SimplexControlador:**
  - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con solucionar un problema de programación lineal.
  - **UML:**



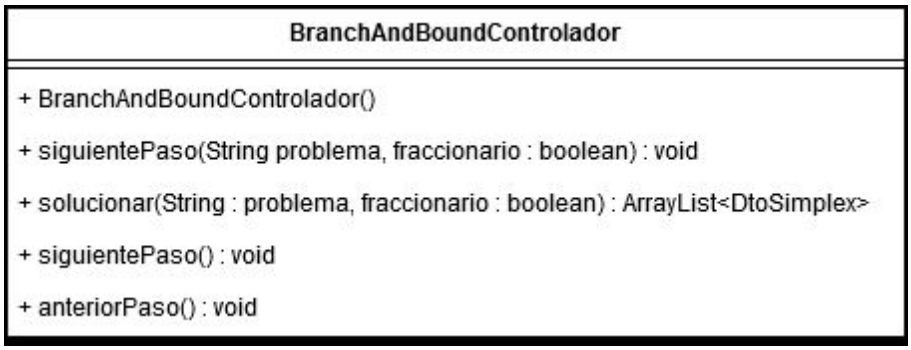
- **MatrizControlador:**
  - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con reconocer una matriz numérica y realizar operaciones pivote en ella.
  - **UML:**



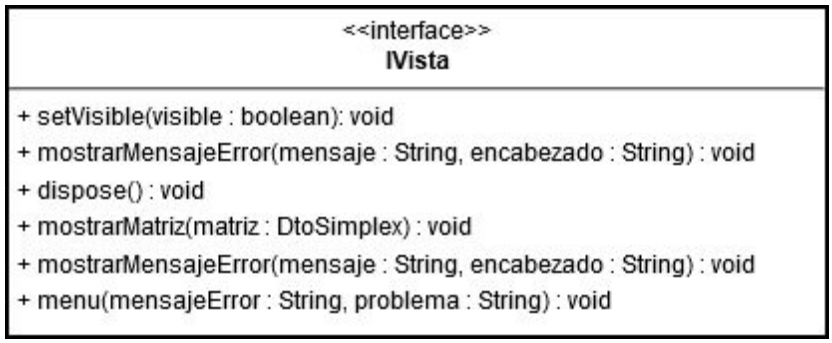
- **GomoryControlador:**
  - **Descripción:** Controlador concreto para el caso de uso Cortes de Gomory.
  - **UML:**



- **BranchAndBoundControlador:**
  - **Descripción:** Controlador concreto para el caso de uso de Branch and Bound.
  - **UML:**



- **IVista:**
  - **Descripción:** Controlador concreto para el caso de uso de Branch and Bound.
  - **UML:**



5.2.3. Vista

Contiene las dos pantallas de interfaces gráficas de la solución: PantallaPrincipal y PantallaPasoIntermedio.

- **PantallaPrincipal:**
  - **Descripción:** Pantalla principal y punto de entrada del programa. Ella es la encargada de recibir el problema de programación lineal por parte del usuario e instanciar el controlador respectivo.
- **PantallaPasoIntermedio:**
  - **Descripción:** Pantalla encargada de mostrar al usuario el estado actual del algoritmo Simplex. Ella tiene lógica para enviarle al controlador las acciones que realiza el usuario, y que de esta manera el controlador actualice la vista de manera respectiva.
- **PantallaPasoIntermedioBranchAndBound:**
  - **Descripción:** Pantalla encargada de mostrar al usuario el estado actual del algoritmo Branch and Bound para soluciones enteras. Implementa la lógica de interacción y el despliegue del árbol textual que representa

5.2.4. Dto

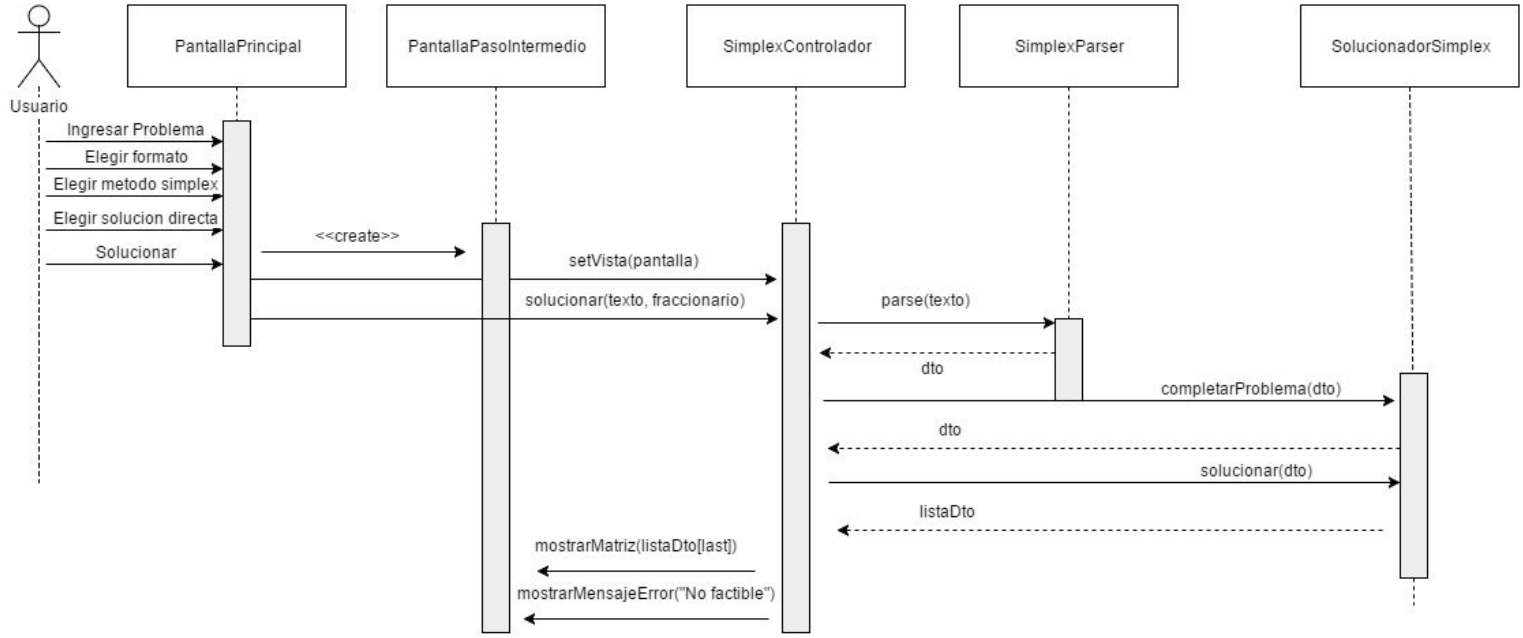
- **Descripción:** Data Transfer Object utilizado para resolver un problema simplex y pasar el resultado entre las diferentes capas arquitecturales.

- **UML:**

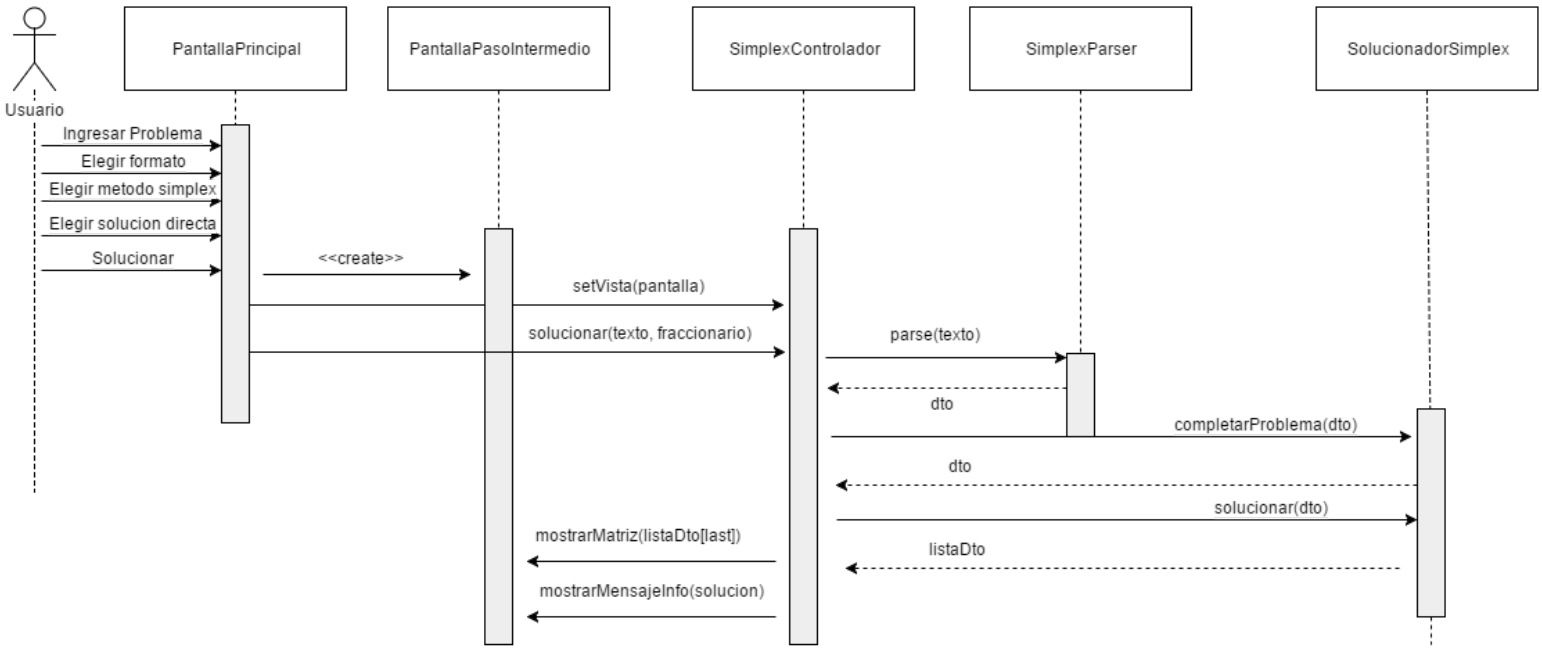
6. Vista de Procesos

Para la vista de proceso, se describirán los casos de uso listados al inicio de este documento, así como los requerimientos funcionales que implican interacción del usuario con el programa por medio de diagramas de interacción para describir la manera en que los objetos se comunican los unos con los otros para realizar las funcionalidades deseadas.

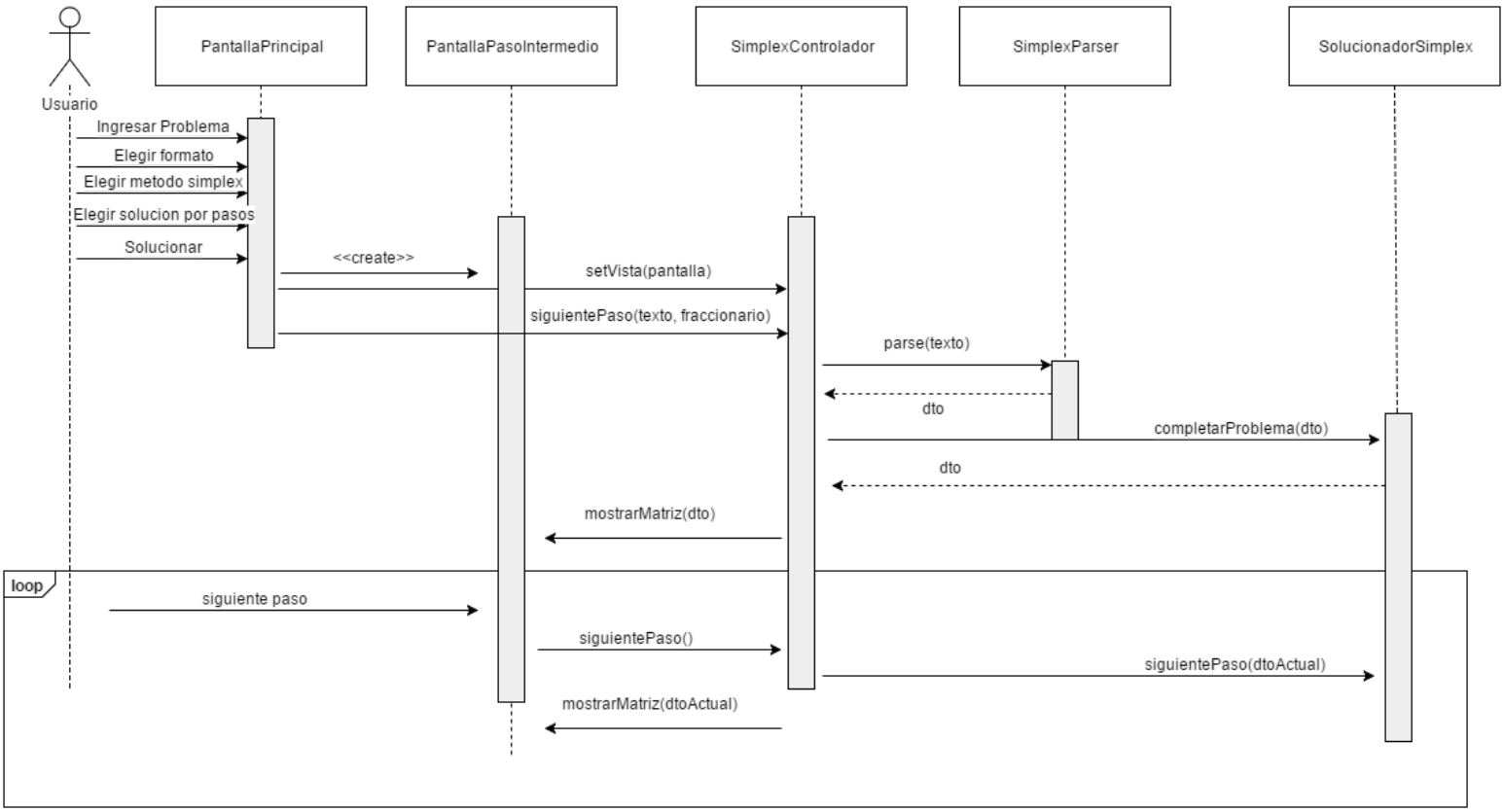
6.1. Verificar la factibilidad de un problema de programación lineal ingresado.



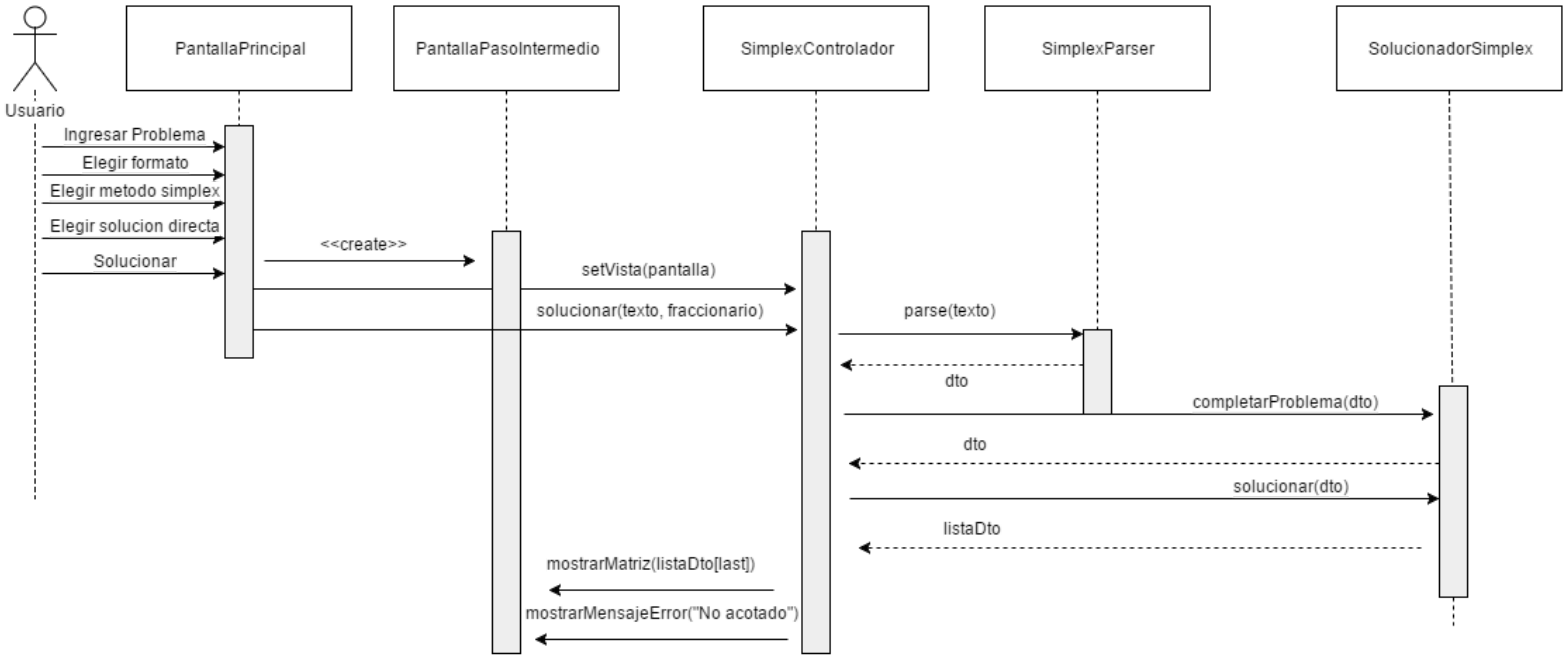
6.2. Obtener de manera inmediata una solución óptima de un problema de programación lineal.



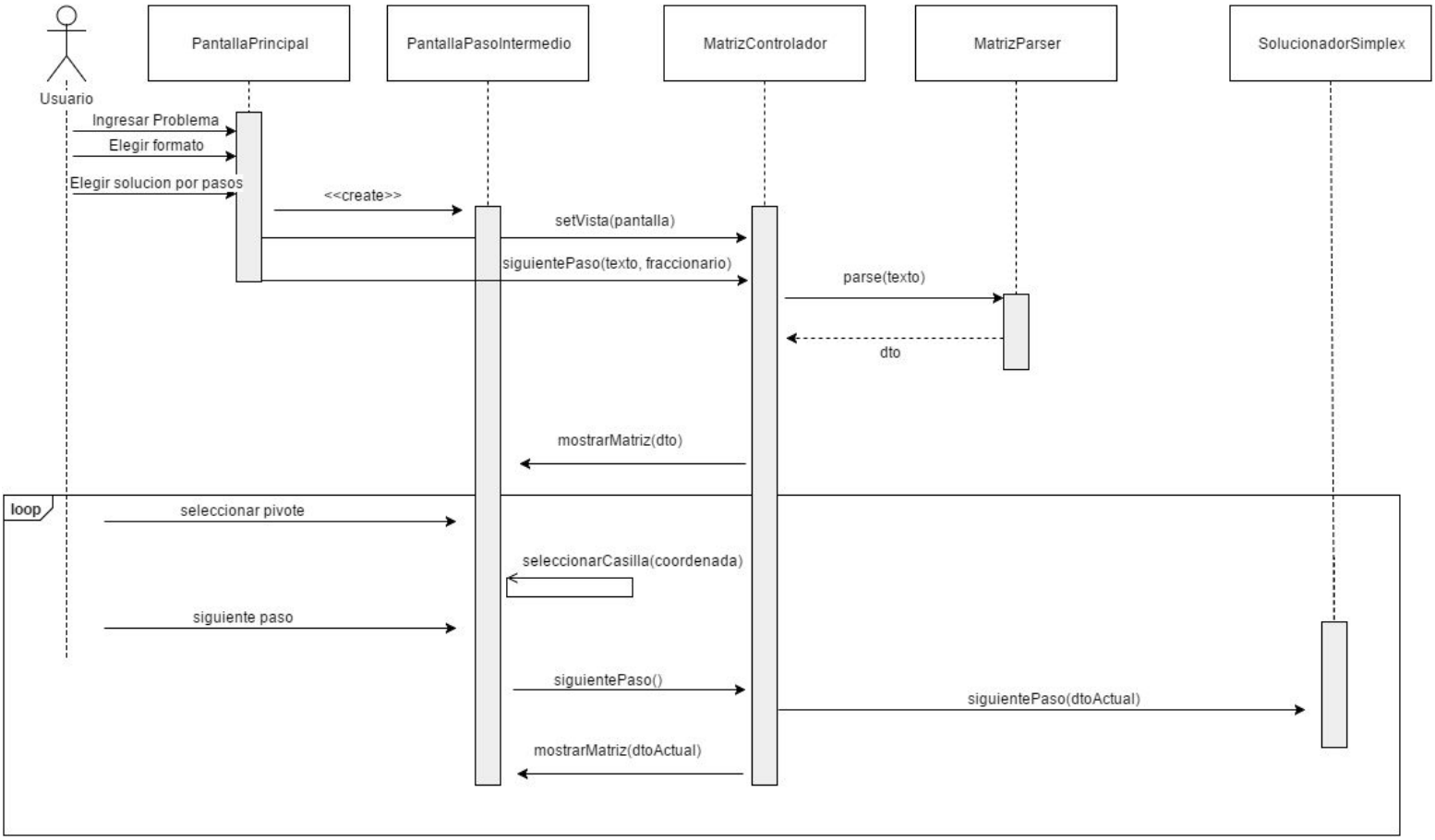
6.3. Listar los pasos intermedios para encontrar la solución a un problema de programación lineal.



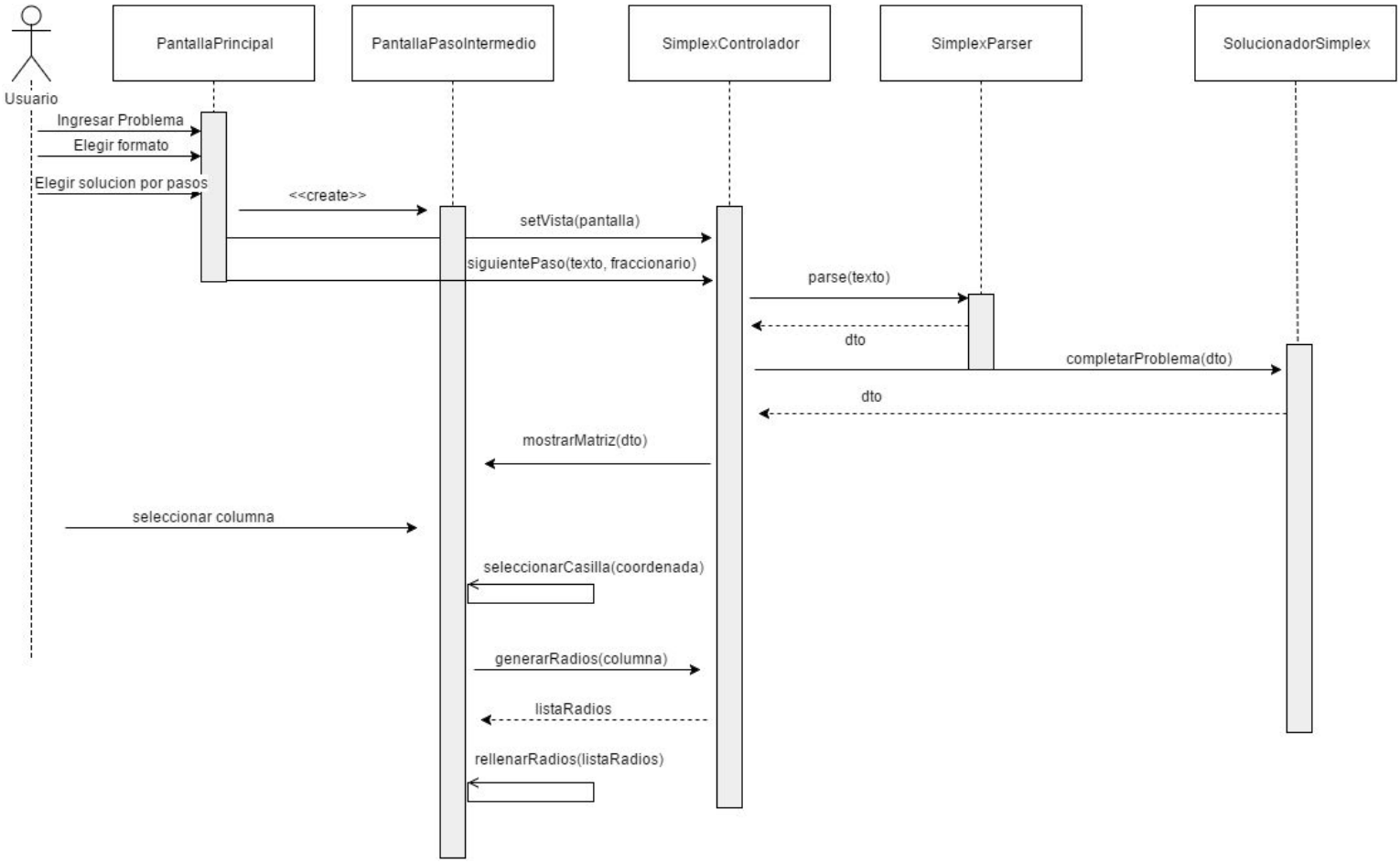
6.4. Verificar si un problema de programación lineal ingresado se encuentra acotado.



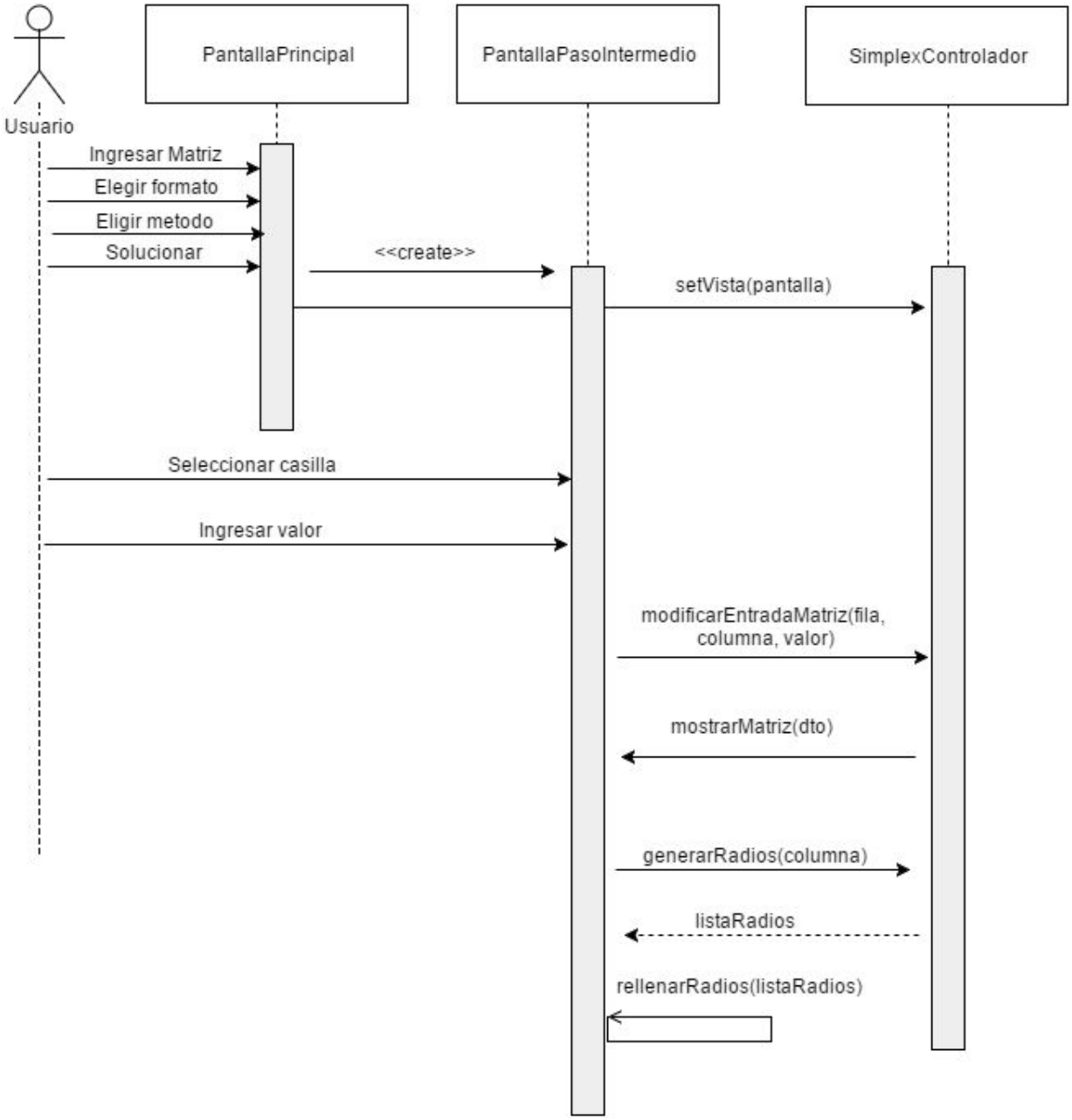
6.5. Ingresar una matriz de N x M manualmente donde se podrá realizar una operación de fila.



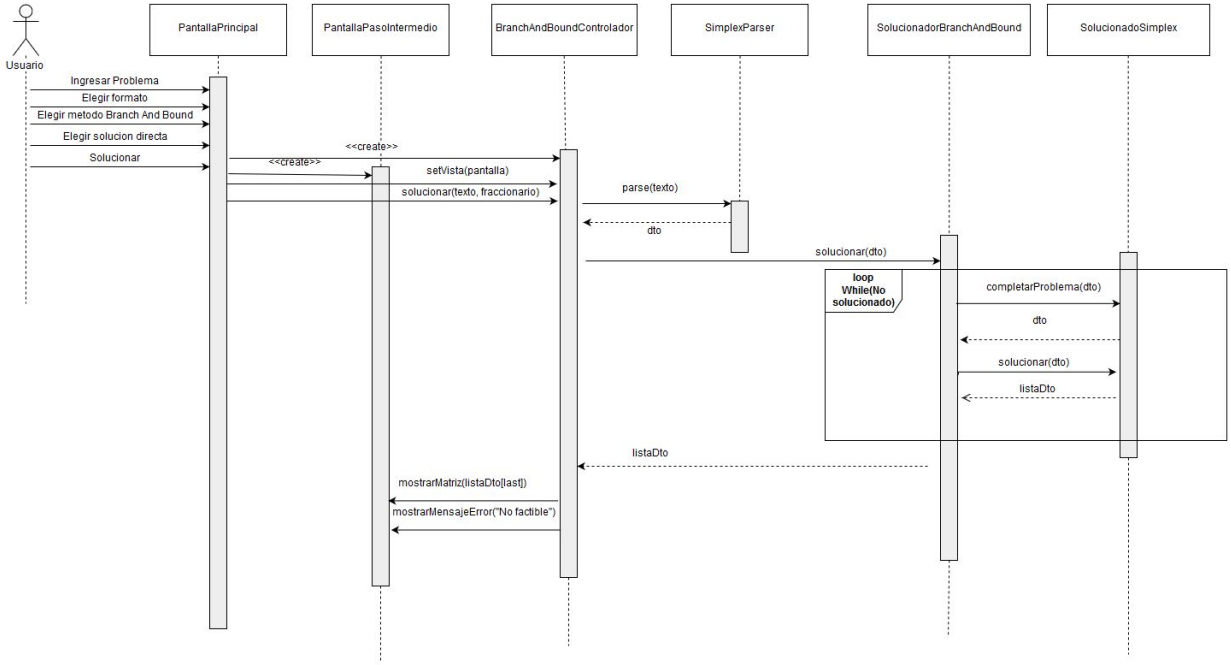
6.6. Obtener los radios entre una columna y el “*lado derecho*” de la matriz



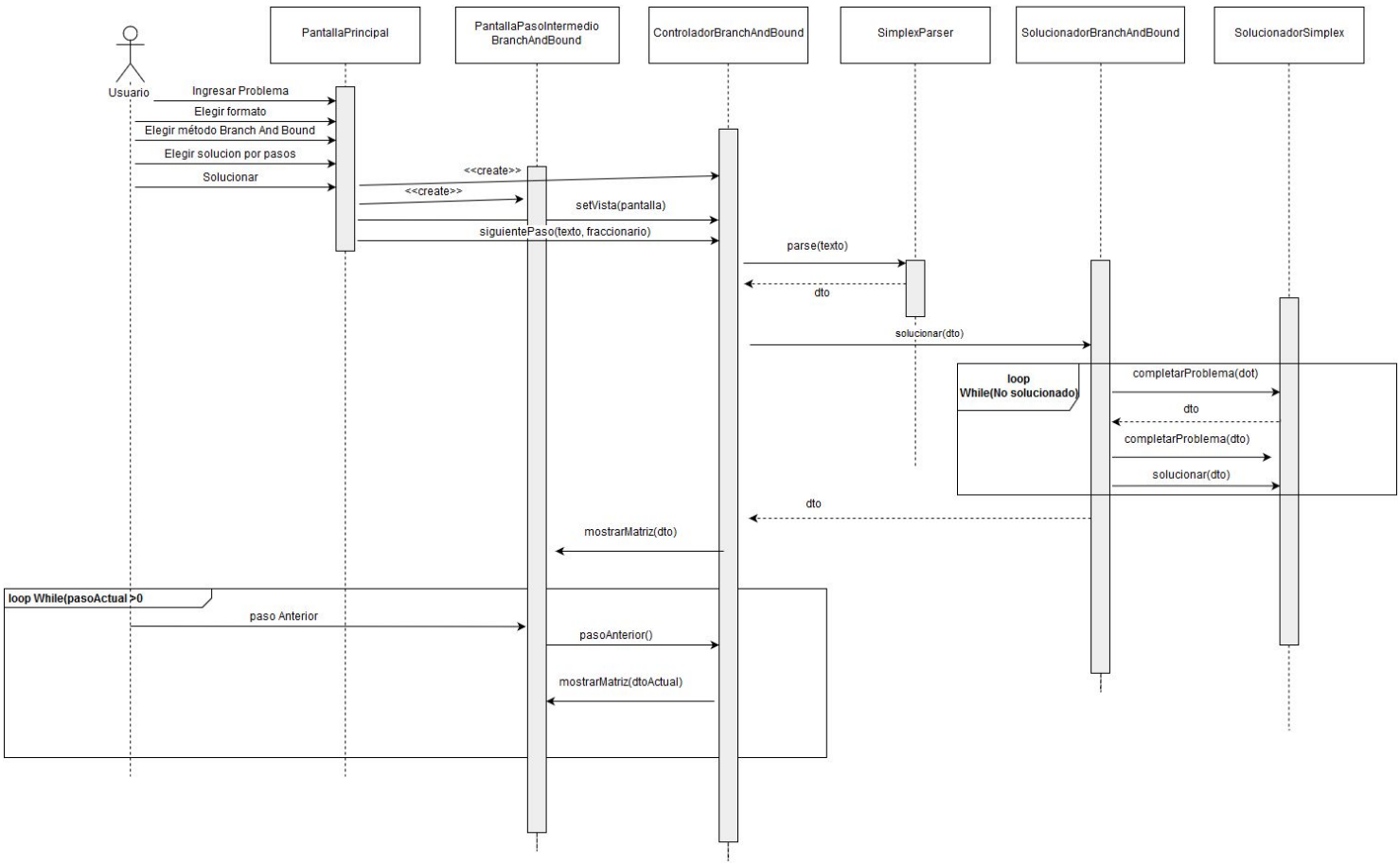
6.7. Modificar la entrada de una matriz



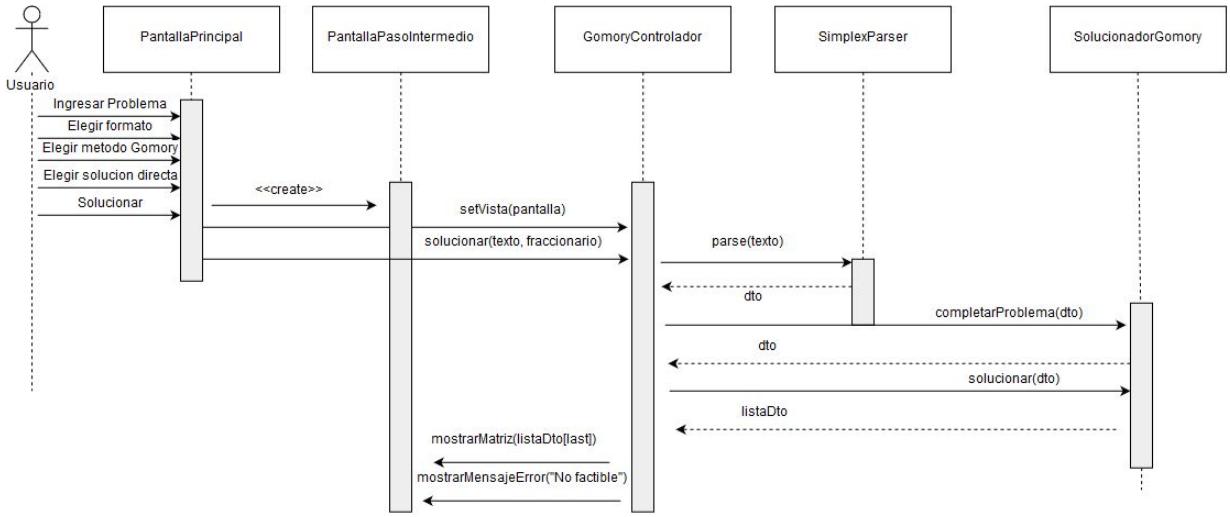
6.8. Solución directa Branch and Bound



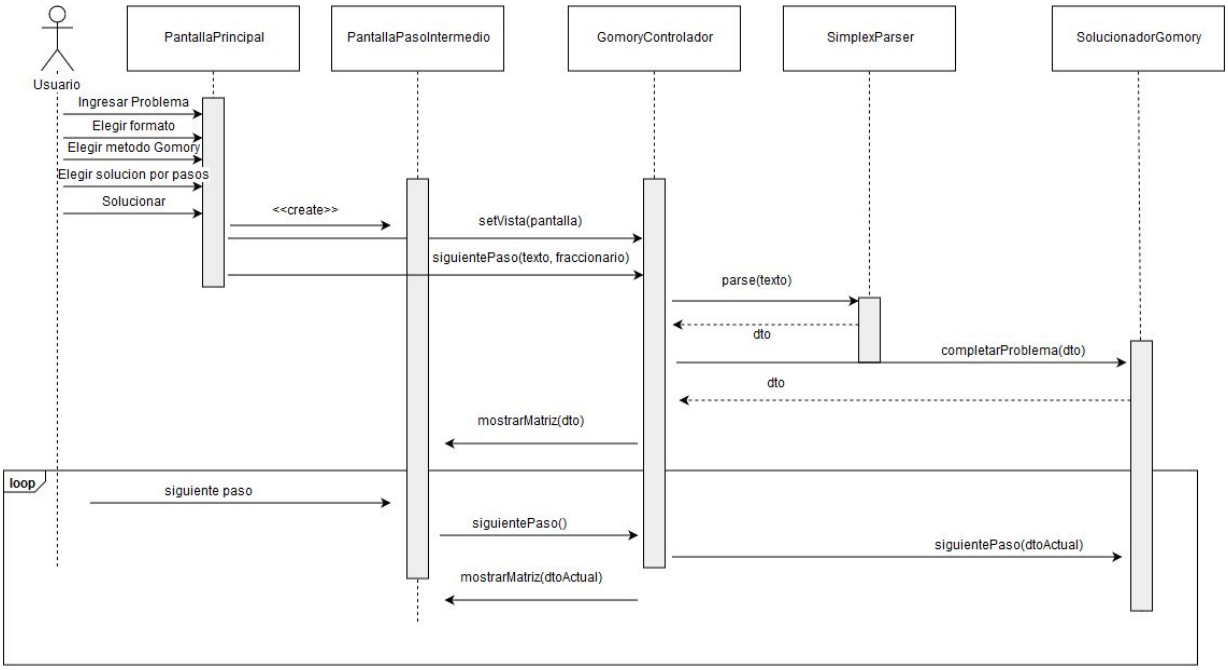
6.9. Solución por pasos Branch and Bound



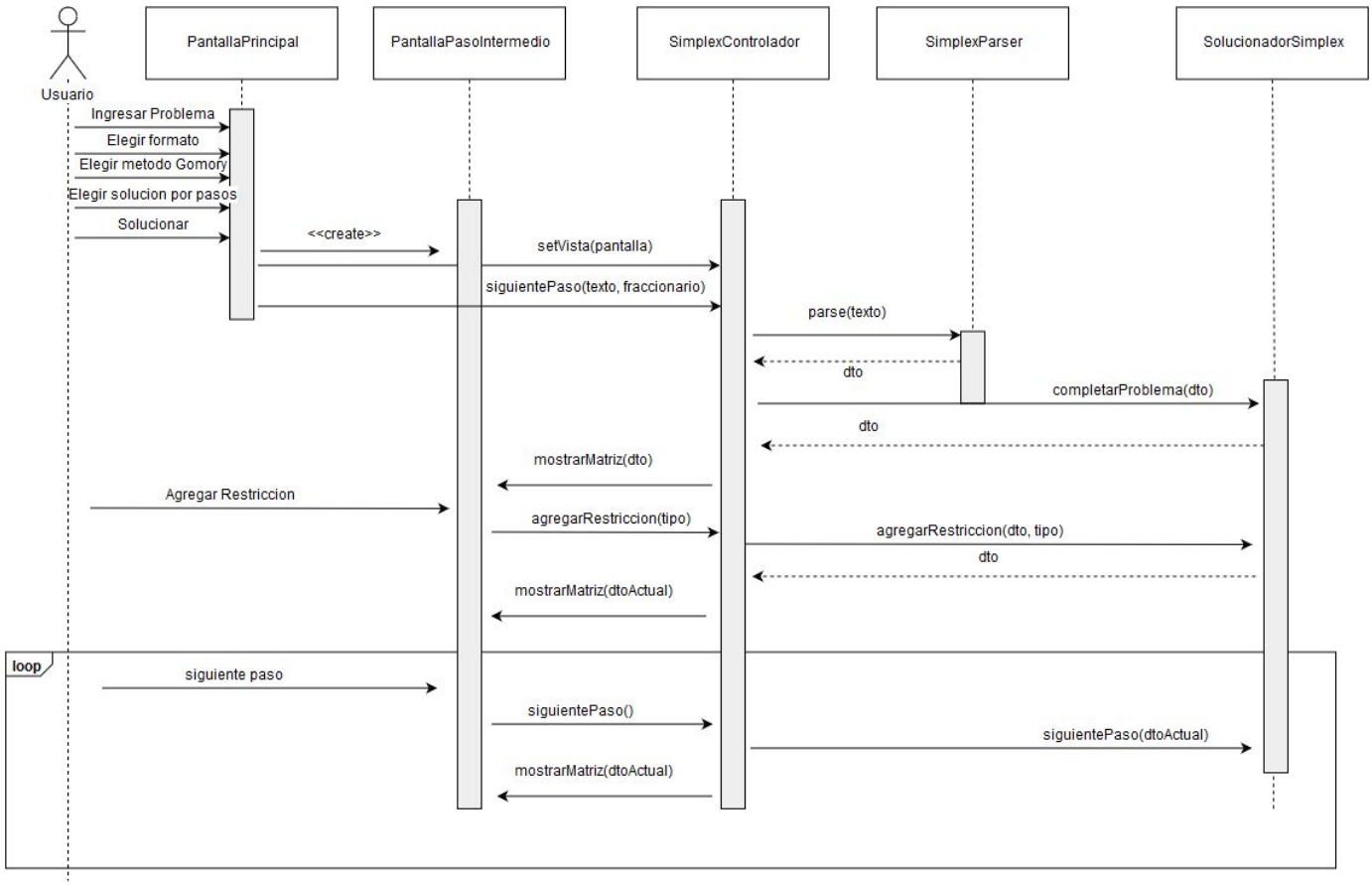
6.10. Solución directa Cortes de Gomory



6.11. Solución por pasos Cortes de Gomory

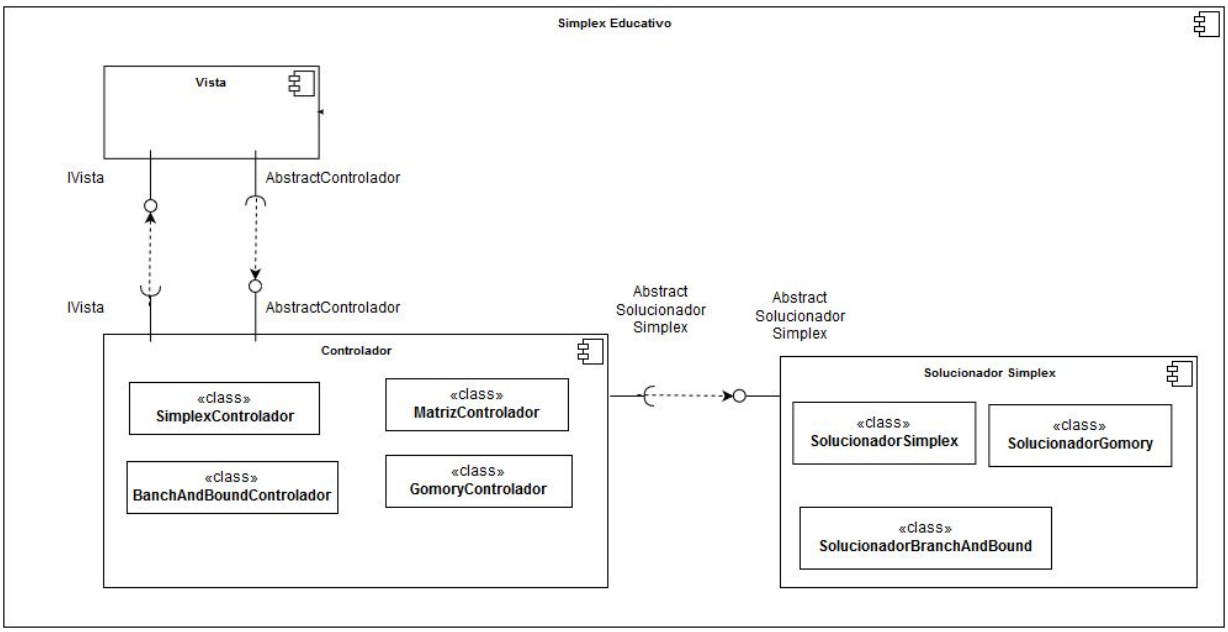


6.12. Agregar restricción a problema.



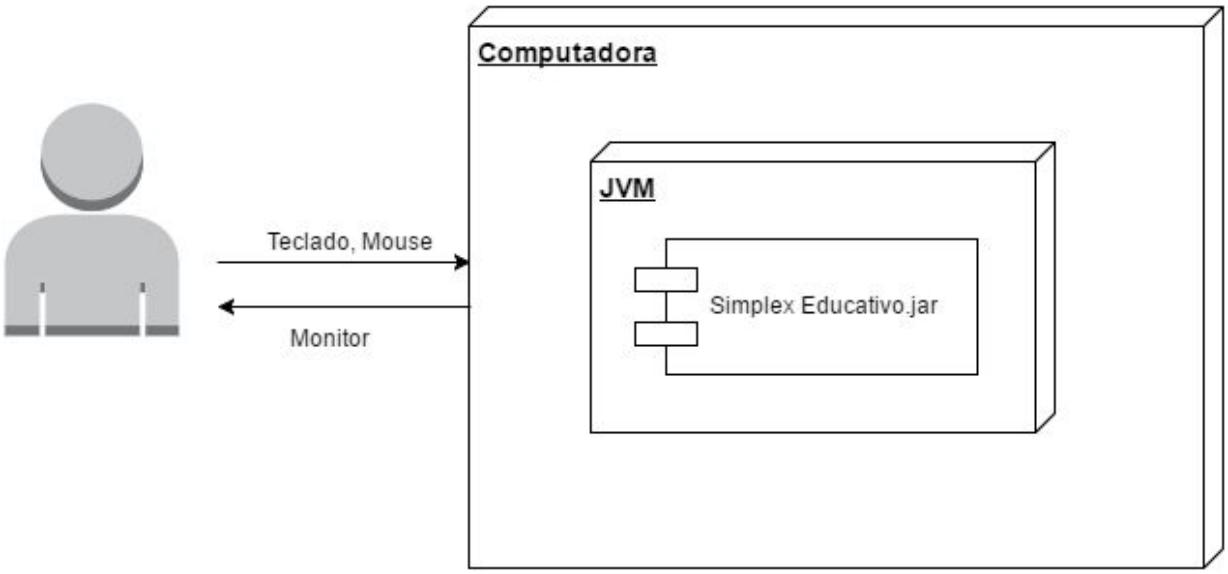
7. Vista de Desarrollo

Dentro de esta sección se describirán, los componentes presentes dentro del sistema, donde se agrupan funcionalidades que posean un fin común dentro de los casos de uso o requerimientos funcionales. A continuación, se encuentra el diagrama de componentes donde se muestran los tres elementos identificados como parte de la solución:



1. **Vista:** contiene la solución gráfica del programa, todos los artefactos pertenecientes a la interfaz gráfica que se le brinda al usuario final se almacenan dentro de este componente. Además se comunica con distintas interfaces del controlador donde se ejecuta validaciones y solicitudes de lógica de negocios para brindar una respuesta representativa a un caso respectivo.
2. **Controlador:** abarca el papel de intermediador, recibe las solicitudes realizadas por el usuario desde la interfaz gráfica, las interpreta y ejecuta la interfaz respectiva de la lógica de negocios con los datos necesarios para retornar un resultado a la solicitud que se realizó.
3. **Solucionador Simplex:** se encarga de la lógica de negocios, recibe la información proveniente del usuario, para luego procesarla y validar el resultado, retornando una solución veraz al caso que se solicitó.
4. **Solucionador Gomory:** Subclase de SolucionadorSimplex. Utiliza métodos de su superclase para realizar su caso de uso. Añade funciones para agregar Cortes de Gomory a un problema de programación lineal basado en una restricción hasta encontrar la solución entera
5. **Solucionador Branch and Bound:** Sublase de SolucionadorSimplex. Utiliza una clase privada para implementar el algoritmo de Branch and Bound, el cual maneja un árbol con las soluciones a problemas intermedios hasta llegar al acoto de una solución óptima.

8. Vista de Despliegue



Dentro del diagrama de despliegue del sistema mostrado anteriormente, se puede observar que solo es constituido por un único nodo, debido a que para ejecutar el programa solo es necesario un equipo de cómputo donde se pueda ejecutar la máquina virtual de Java. También es importante agregar que no existe ninguna dependencia a un servidor, base de datos o servicio web, por esta razón no se agrega ningún otro dispositivo que represente a estos elementos dentro del diagrama.

9. Vista de Datos

No se han detectado requerimientos de persistencia de datos ni afines.

Anexos

Anexo A: Parseo del texto de entrada

Scanner

Para la realización del scanner que divida la cadena de texto en los diferentes tipos de tokens reconocibles se utilizó la herramienta JFlex. Esta herramienta permite definir una serie de expresiones regulares y generar código que deba ser ejecutado cuando el scanner encuentre dicha expresión regular en el texto que está analizando. Las expresiones regulares se definieron en el archivo /Implementacion/modelo/parser/Scanner.flex. Dicho archivo puede ser compilado utilizando jflex-1.6.1.jar mediante el siguiente comando:

java -jar jflex-1.6.1.jar Scanner.flex

Dicho comando creará la clase Scanner.java que puede ser luego agregada al proyecto y compilada junto al resto. La documentación de JFlex puede ser accedida en el siguiente enlace: http://jflex.de/manual.html.

Parser

Para la realización del parser se utilizó la herramiento CUP. Esta herramienta trabaja en conjunto con CUP para lograr un análisis léxico y sintáctico de una cadena de texto. Al igual que JFlex, la gramática se define en un archivo aparte que luego puede ser compilado utilizando un .jar que generará código Java que luego puede ser compilado utilizando el compilador de java. El parser posee una referencia al scanner generado por JFlex, e invoca a su método next_token() cada vez que necesita el próximo token del string. La gramática se define en el archivo /Implementacion/modelo/parser/Parser.cup y puede ser compilada mediante el siguiente comando:

java -jar java-cup-11b.jar package "modelo.parser" -parser "SimplexParser" -Rules.cup

El comando anterior creará dos clases: SimplexParser.java y sym.java. Ambas deben ser copiadas al proyecto y compiladas junto con el resto de archivos fuente. La documentación de cup puede ser accedida en el siguiente enlace: http://www2.cs.tum.edu/projekte/cup/manual.html