Documento de Arquitectura de Software

Profesora: Maria Estrada.

Curso: Proyecto de Ingeniería de Software.

18.01.2017

Jose Fernando Molina Chacón

Yordan Jiménez Hernández

Cambios

Descripción	Versión	Fecha
Versión Inicial	0.1	6/01/17
Agregados casos de uso de iteración 2.	0.2	11/01/17
Agregados casos de uso de iteración 3.	0.3	18/01/17

Contenidos

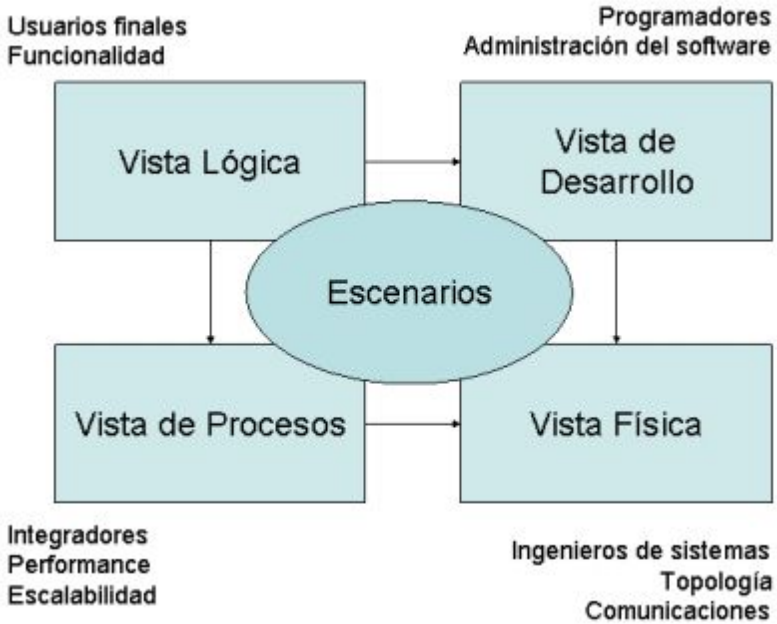
1. Introducción

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 construido 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.

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 físico 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

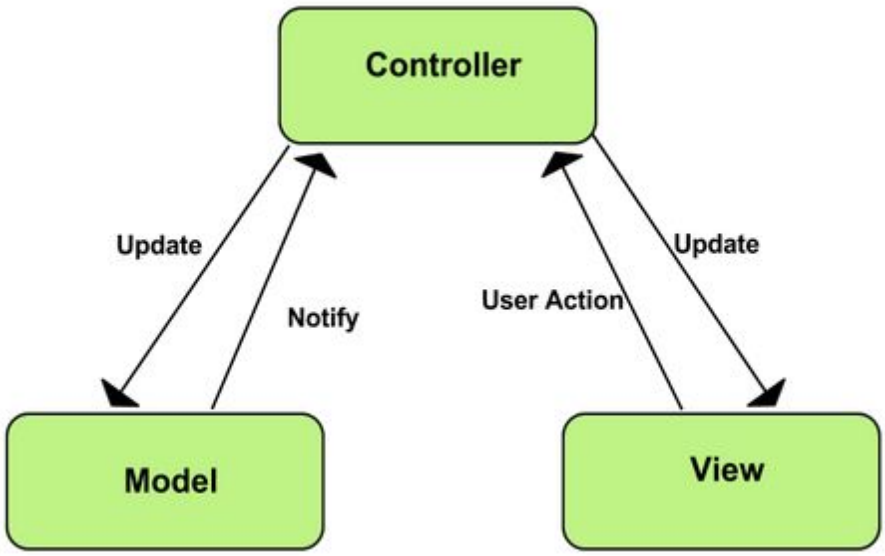
Para el diseño arquitectural se debe tomar en cuenta los principios SOLID para diseño de software. Dichos principios se describen a continuación:

- **Single Responsibility 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	Existen ciertos problemas de programación lineal que, debido a las restricciones que los definen, es matemáticamente imposible cumplir con todas las restricciones impuestas. Por ejemplo, que una restricción defina la variable x1 <= 4 y otra defina x1>=5. El sistema debe poseer la capacidad para determinar si un problema de programación lineal posee una solución factible óptima. Para ello, podrá reconocer una cadena de texto ingresada por el usuario con un formato establecido (el formato utilizado por el profesor Jose Helo en sus apuntes de clase) y extraer de él la información necesaria para representar el problema en forma tabular y, por medio del algoritmo Simplex, determinar si existe una solución óptima factible.

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	Los problemas de programación lineal buscan maximizar o minimizar una función objetivo con N variables lineales sujetas a M inecuaciones lineales que incluyen las variables de la función objetivo. El método Simplex es un algoritmo desarrollado por George Dantzig para resolver esta clase de problemas. Ellos se utilizan en diversos campos de la matemática aplicada e ingenierías para encontrar soluciones a problemas que puedan ser modelados mediante el método de programación lineal. El sistema será capaz de solucionar un problema de programación lineal con N variables y M restricciones lineales por medio del método Simplex, brindando inmediatamente la solución al usuario.

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 algoritmo simplex representa matricialmente (esto es, por medio de una matriz numérica) el problema de programación lineal de modo que cada columna representa una variable y cada fila representa una restricción. Por medio de operaciones de pivote (esto significa escoger una entrada de la matriz, convertir en 1 dicha entrada y realizar operaciones fila para reducir las otras entradas de la columna a 0), el algoritmo se traslada entre las posibles soluciones hasta encontrar una solución óptima. En cada paso del algoritmo, hay una operación de pivote que cambia el estado de la matriz que representa el problema lineal. El sistema debe ser capaz de solucionar un problema de programación lineal por medio del método Simplex donde se pueda generar las tablas o matrices intermedias que se produjeron para obtener la primera solución óptima del problema ingresado. Esto significa que debe ser posible

	observar todas las operaciones de pivote realizadas para llegar a la solución óptima encontrada.

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	Existen ciertos problemas de programación lineal que, debido a las restricciones que los definen, es matemáticamente imposible llegar a un máximo o un mínimo absoluto. Por ejemplo, si se buscara maximizar la funcion f(x1,x2) = x1 + x2, sujeto solamente a la restricción x1 <= 4, entonces estamos ante un problema no acotado. Esto significa que no hay un límite para maximizar la función f(x1,x2), pues la variable x2 no posee un límite (acote) bajo el cual se deba mantener, por lo que la función puede ser maximizada infinitamente. El sistema debe poseer la capacidad para determinar si un problema de programación lineal posee una solución acotada, esto es, posee un límite superior o inferior. El algoritmo simplex explica cuándo un problema no está acotado, esto sucede cuando los radios (división entre el lado derecho de la matriz y la columna sobre la cual se busca pivotar) que se utilizan para elegir la fila pivote son todos infinito, significando que la función objetivo puede crecer de manera infinita sobre todas las restricciones.

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	Debido a que en esencia el algoritmo simplex trabaja con operaciones de matrices, el cliente desea poseer la capacidad de ingresar manualmente una matriz de N x M números en formato fraccionario o decimal y escoger manualmente la entrada sobre la cual se debe pivotar. También deben mostrarse los radios entre el lado derecho de la matriz (esto es, la última columna) y la columna que está seleccionada en cierto momento. El usuario será el encargado de escoger en cada paso la entrada sobre la cual desea pivotar, y el sistema no hará ninguna asunción o sugerencia acerca de la próxima entrada que debería ser escogida en cada paso.

4.1.6. Obtener de manera inmediata la solución entera de un problema de programación lineal mediante el algoritmo de Gomory.

Nombre	Obtener de manera inmediata la solución entera de un problema de programación lineal mediante el algoritmo de Gomory.
Actor	Usuario Final
Sinopsis	El método Simplex para resolución de problemas de programación lineal encuentra soluciones factibles en el dominio de los números reales. Esto significa que es posible encontrar respuestas que no sean enteras, osea, fracciones y decimales pueden ser el valor que tomen las variables o el resultado de la función objetivo. Sin embargo, en algunos casos es necesario encontrar la solución entera a un problema de programación lineal por su naturaleza, debido a que puede ser el caso de que las variables representan objetos de la vida real que no puedan ser subdivididos en partes fraccionarias. Para resolver el problema de encontrar la solución entera óptima existen varios métodos, uno de ellos es el denominado algoritmo de Cortes de Gomory, en el cual se resuelve un problema de programación lineal mediante el

	Simplex común y, en caso de llegar a una solución óptima no entera, se agrega una restricción extra al problema (denominado “corte”) y se vuelve a resolver el problema. Este método se repite N veces hasta encontrar una respuesta con todas las variables y el resultado de la función objetivo enteras. Debe ser posible encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de cortes de gomory de manera inmediata, esto es, sin mostrar los pasos intermedios. Una vez encontrada la solución, el usuario podrá observar el resumen de pasos o la tabla final, siendo posible devolverse en la lista de soluciones.

4.1.7. Listar los pasos intermedios para encontrar la solución de un problema de programación lineal mediante el algoritmo de Gomory

Nombre	Listar los pasos intermedios para encontrar la solución de un problema de programación lineal mediante el algoritmo de Gomory
Actor	Usuario Final
Sinopsis	Muy relacionado al caso de uso anterior. Debe ser posible listar los pasos intermedios para encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de cortes de gomory. El sistema debe ir indicando paso por paso el estado actual del problema por medio de mensajes y notificar cuándo se ha llegado un óptimo no entero, junto con el corte agregado y la fila que fue escogida para realizar el corte.

4.1.8. Obtener de manera inmediata una solución entera de un problema de programación lineal mediante el algoritmo de Branch and Bound.

Nombre	Obtener de manera inmediata una solución entera de un problema de programación lineal mediante el algoritmo de Branch and Bound.
Actor	Usuario Final.
Sinopsis	La meta de este caso de uso es la misma del caso de uso anterior, llegar a la solución entera óptima para un problema de programación lineal, pero esta vez se debe utilizar el algoritmo de Branch and Bound. Este algoritmo es parecido a los Cortes de Gomory debido a que también agrega restricciones al final del problema, pero se diferencia en que utiliza un árbol de soluciones que va poco a poco siendo construido por medio de restricciones mayor o igual y menor o igual. Dicho árbol es generado hasta encontrar una solución entera óptima. Debe ser posible encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de Branch and Bound de manera inmediata. Dicho algoritmo resuelve N cantidad de subproblemas simplex agregando restricciones según sea necesario en un paso dado, llegando a acotar la solución en algún momento al valor óptimo válido.

4.1.9. Listar los pasos intermedios para encontrar la solución de un problema de programación lineal mediante el algoritmo de Branch and Bound.

Nombre	Listar los pasos intermedios para encontrar la solución de un problema de programación lineal mediante el algoritmo de Branch and Bound.
Actor	Usuario Final.
Sinopsis	La meta de este caso de uso es la misma del caso de uso Cortes de Gomory, llegar a la solución entera óptima para

	un problema de programación lineal, pero esta vez se debe utilizar el algoritmo de Branch and Bound. Este algoritmo es parecido a los Cortes de Gomory debido a que también agrega restricciones al final del problema, pero se diferencia en que utiliza un árbol de soluciones que va poco a poco siendo construido por medio de restricciones mayor o igual y menor o igual. Dicho árbol es generado hasta encontrar una solución entera óptima. Debe ser posible encontrar la solución entera óptima de un problema de programación lineal por medio del algoritmo de Branch and Bound de manera inmediata. Dicho algoritmo resuelve N cantidad de subproblemas simplex agregando restricciones según sea necesario en un paso dado, llegando a acotar la solución en algún momento al valor óptimo válido.

4.1.10. Agregar restricciones a un problema de programación lineal.

Nombre	Agregar restricciones a un problema de programación lineal.
Actor	Usuario Final.
Sinopsis	El usuario desea poder agregar manualmente restricciones una vez que se muestra el problema en su representación matricial. Esto se debe a que el usuario desea manualmente agregar cortes de gomory o restricciones de Branch and Bound manualmente como técnica educativa. Debe ser posible, una vez iniciado el programa, agregar restricciones a la matriz que representa el programa de programación lineal. Dichas restricciones pueden ser de tipo mayor igual, menor igual o igual. En el caso de menor o igual, se agregará una nueva variable de holgura. En el caso de igual, se agregará una nueva variable artificial. En el caso de mayor o igual, se agregará una variable negativa de holgura y una artificial.

4.1.11. Obtener de manera inmediata una solución óptima de un problema de programación lineal. (Caso móvil)

Nombre	Agregar restricciones a un problema de programación lineal.
Actor	Usuario Final.
Sinopsis	Los problemas de programación lineal buscan maximizar o minimizar una función objetivo con N variables lineales sujetas a M inecuaciones lineales que incluyen las variables de la función objetivo. El método Simplex es un algoritmo desarrollado por George Dantzig para resolver esta clase de problemas. Ellos se utilizan en diversos campos de la matemática aplicada e ingenierías para encontrar soluciones a problemas que puedan ser modelados mediante el método de programación lineal. la aplicación móvil será capaz de solucionar un problema de programación lineal con N variables y M restricciones lineales por medio del método Simplex, brindando inmediatamente la solución al usuario.

4.1.12. Obtener de manera inmediata una solución óptima de un problema de programación lineal. (Caso móvil)

Nombre	Agregar restricciones a un problema de programación lineal.
Actor	Usuario Final.
Sinopsis	Los problemas de programación lineal buscan maximizar o minimizar una función objetivo con N variables lineales sujetas a M inecuaciones lineales que incluyen las

	variables de la función objetivo. El método Simplex es un algoritmo desarrollado por George Dantzig para resolver esta clase de problemas. Ellos se utilizan en diversos campos de la matemática aplicada e ingenierías para encontrar soluciones a problemas que puedan ser modelados mediante el método de programación lineal. La aplicación móvil será capaz de solucionar un problema de programación lineal con N variables y M restricciones lineales por medio del método Simplex, brindando inmediatamente la solución al usuario.

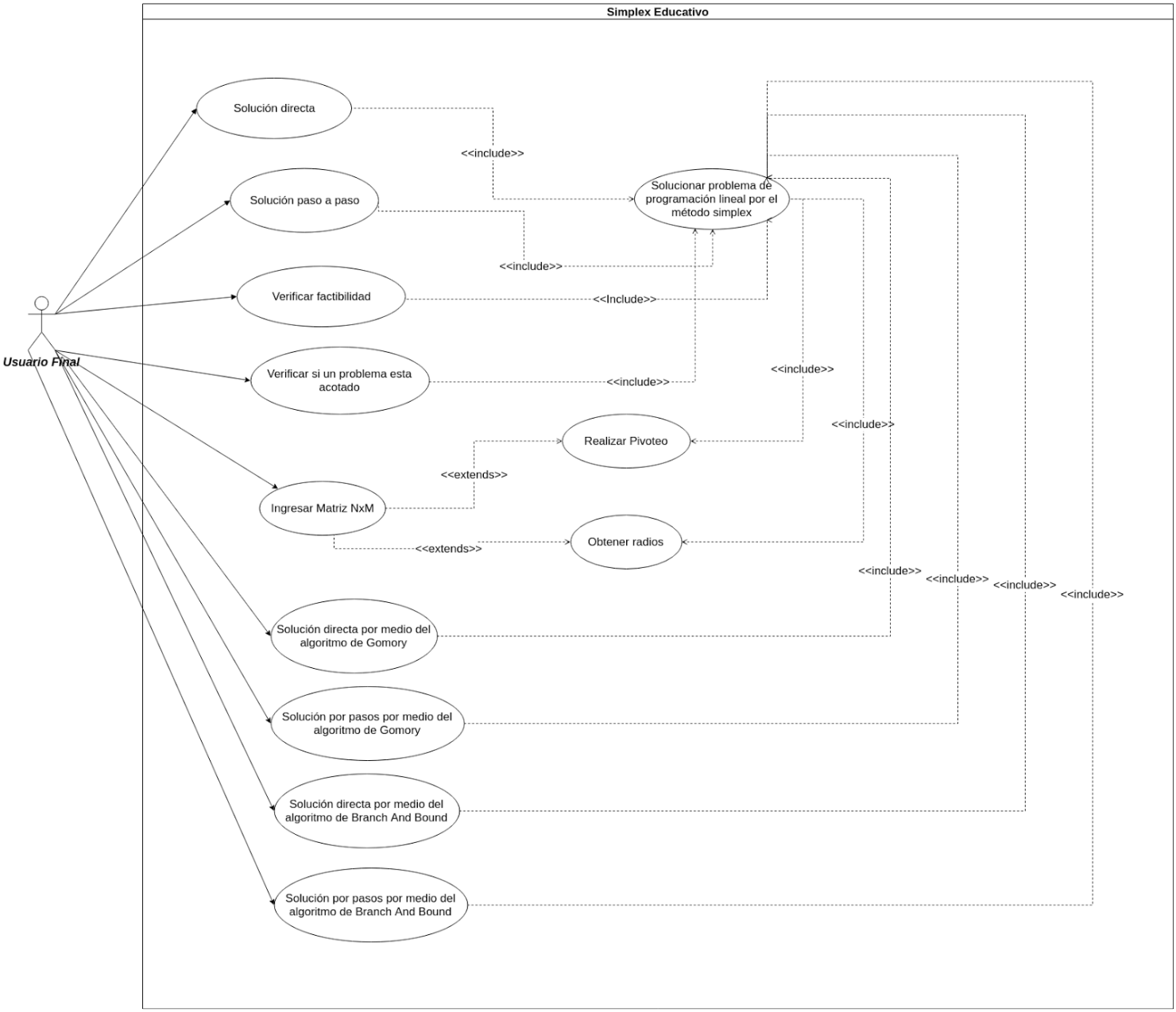
4.1.13. Obtener de manera inmediata una solución óptima de un problema de programación lineal. (Caso móvil)

Nombre	Agregar restricciones a un problema de programación lineal.
Actor	Usuario Final.
Sinopsis	Los problemas de programación lineal buscan maximizar o minimizar una función objetivo con N variables lineales sujetas a M inecuaciones lineales que incluyen las variables de la función objetivo. El método Simplex es un algoritmo desarrollado por George Dantzig para resolver esta clase de problemas. Ellos se utilizan en diversos campos de la matemática aplicada e ingenierías para encontrar soluciones a problemas que puedan ser modelados mediante el método de programación lineal. La aplicación móvil será capaz de solucionar un problema de programación lineal con N variables y M restricciones lineales por medio del método Simplex, brindando inmediatamente la solución al usuario.

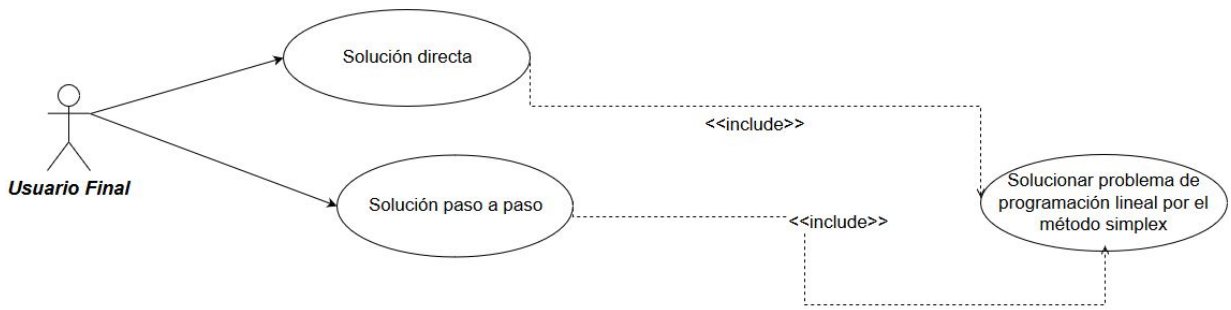
4.1.14. Listar los pasos intermedios para encontrar la solución a un problema de programación lineal. (Caso móvil)

Nombre	Agregar restricciones a un problema de programación lineal.
Actor	Usuario Final.
Sinopsis	El algoritmo simplex representa matricialmente (esto es, por medio de una matriz numérica) el problema de programación lineal de modo que cada columna representa una variable y cada fila representa una restricción. Por medio de operaciones de pivote (esto significa escoger una entrada de la matriz, convertir en 1 dicha entrada y realizar operaciones fila para reducir las otras entradas de la columna a 0), el algoritmo se traslada entre las posibles soluciones hasta encontrar una solución óptima. En cada paso del algoritmo, hay una operación de pivote que cambia el estado de la matriz que representa el problema lineal. La aplicación debe ser capaz de solucionar un problema de programación lineal por medio del método Simplex donde se pueda generar las tablas o matrices intermedias que se produjeron para obtener la primera solución óptima del problema ingresado. Esto significa que debe ser posible observar todas las tablas intermedias que fueron iteradas en cada paso para llegar a la solución final.

4.2. Diagrama de Casos de Uso para escritorio



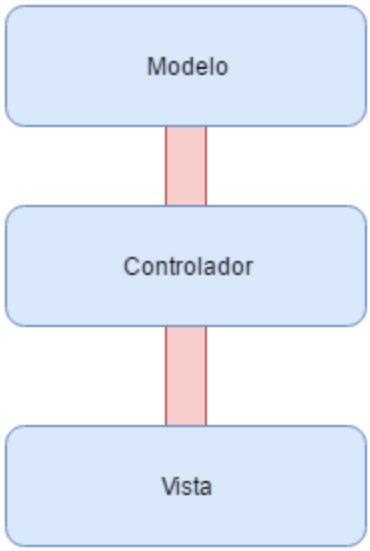
4.3. Diagrama de casos de uso para versión móvil



5. Vista logica

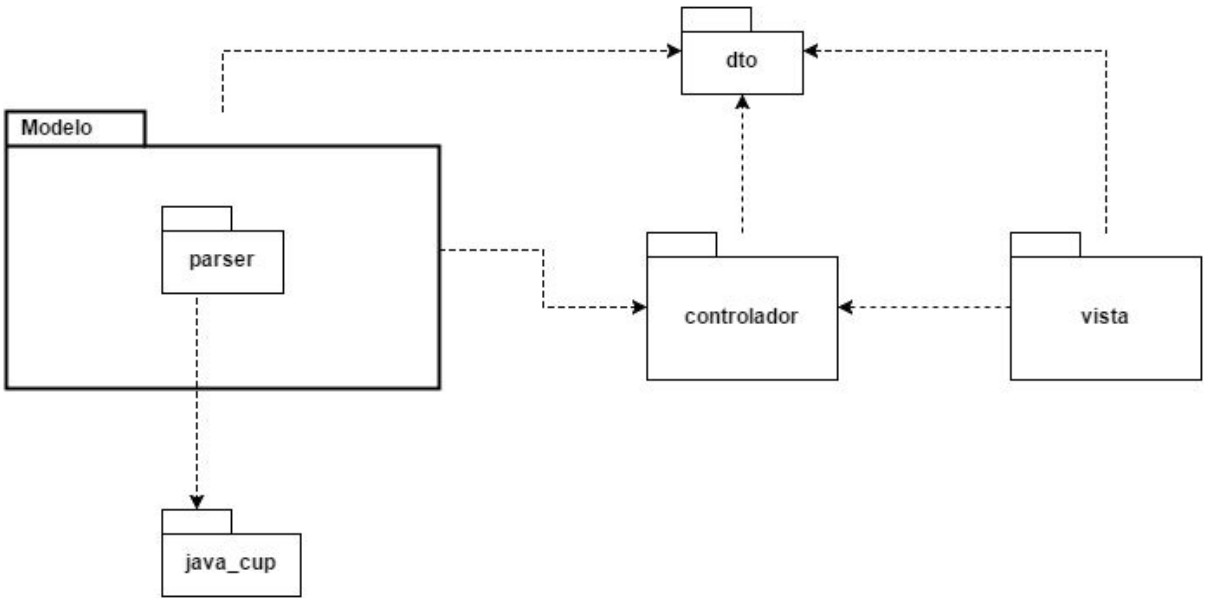
5.1. Vista general

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.



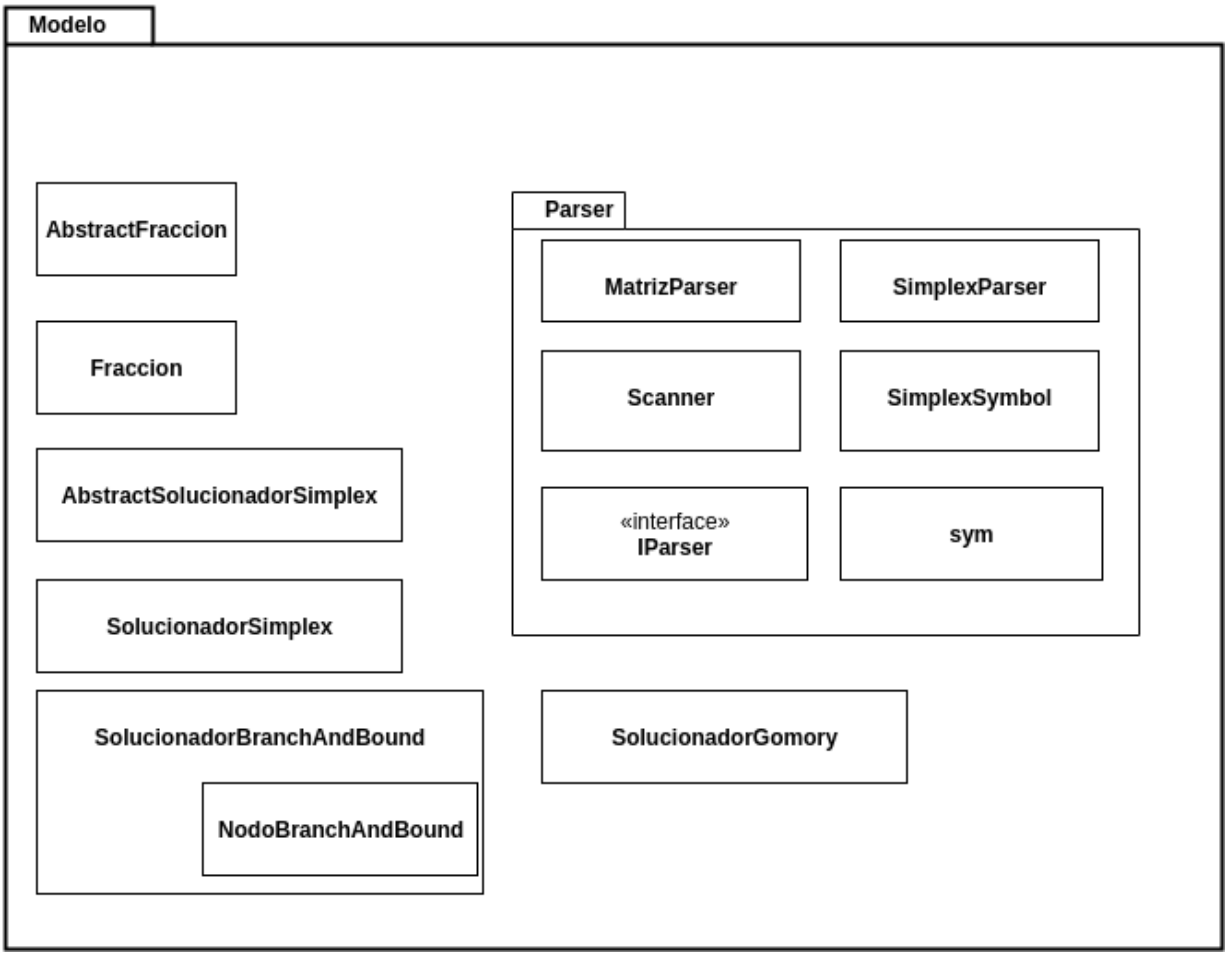
- **Modelo:** Es la capa encargada de representar y manejar el comportamiento de los datos solamente. Ella responde a estímulos por parte de la capa controlador únicamente. Es en esta capa donde se encuentran todas las clases necesarias para la ejecución de los casos de uso a nivel lógico meramente, 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 responsable de proveer las funcionalidades necesarias para resolver los casos de uso descritos anteriormente, siendo como objeto principal solucionar problemas de programación lineal por medio del método Simplex.



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:**

- **Atributos:**
  - **Numerador:** representa la parte del numerador de una fracción matemática.
  - **Denominador:** representa el denominador de una fracción matemática, por defecto posee un valor de 1.
- **Constructores:**
  - **AbstractFraccion(numerador:int,denominador:int) :** Inicializa una instancia de fracción asignando los valores de denominador y numerador indicados por medio de parámetros, puede lanzar una excepción si el denominador es 0.
  - **AbstractFraccion(numerador:double) :** inicializa una instancia de fracción convirtiendo un valor double a su valor fraccional, este resultado se

obtiene contando la cantidad n de decimales que posee el parámetro y creando una fracción con numerador el valor entero que representa con los decimales y denominador con un valor de 10^n cantidad de decimales.

- **Métodos:**
  - **sumar(operando : AbstractFraccion) :** realiza la operación de suma de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:

(Numerador1 * denominador2 **+** Numerador2*Denominador1) **/** (Denomindor1*Denominador2)

Retornando una nueva instancia de Fraccion.

  - **restar(operando : AbstractFraccion) :** realiza la operación de resta de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:

(Numerador1 * denominador2 **-** Numerador2*Denominador1) **/** (Denomindor1*Denominador2)

Retornando una nueva instancia de Fraccion.

  - **multiplicar(operando : AbstractFraccion) :** realiza la operación de multiplicación de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:

(Numerador1 * Numerador2*) **/** (Denomindor1*Denominador2)

Retornando una nueva instancia de Fraccion.

  - **Dividir(operando : AbstractFraccion) :** realiza la operación de división de fracciones matemáticas, tomando como primer operador la fracción donde se llame el método y como segundo operando la fracción indicada por medio de parámetro. La operación se realiza por medio del siguiente método:

(Numerador1 * Denominador2*) / (Denomindor1*Numerador2)

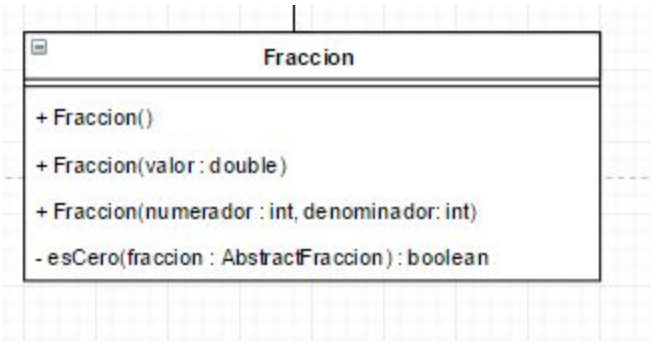
Retornando una nueva instancia de Fraccion.

- **clonar() :** crea una nueva referencia o instancia del objeto tipo Fraccion donde se invoque este método. El numerador y denominador se mantienen igual a la instancia original.
- **obtenerInverso() :** crea una nueva referencia o instancia del objeto tipo Fraccion donde se invoque este método, creando el inverso multiplicativo. El numerador y denominador cambian el papel, en otras palabras en la nueva instancia el numerador de la fracción original ahora sería el denominador y el denominador sería el numerador.
- **obtenerParteDecimal() :** Retorna la parte no entera del número real que representa la fracción. Dicho resultado es encapsulado en una nueva Fraccion con el numerador siendo la parte decimal y con un denominador de 1.
- **obtenerParteEntera() :** Retorna la parte entera del número real que representa la fracción. Dicho resultado es encapsulado en una nueva Fraccion con el numerador siendo la parte entera y con un denominador de 1.
- **toString() :** retorna un valor String de la Fraccion, con el siguiente formato que lo representa "*numerador / denominador*".
- **toString(fraccional:boolean) :** retorna un valor String de la Fraccion,con un formato indicado, si el parámetro es **true** se creará un string con el formato de la función toString() indicada anteriormente y si es **false** retorna un String con el formato de un número real con coma o punto que representa la fracción.
- **iguales(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **igual** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que son iguales y **false** lo contrario.
- **mayorIgualQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **mayor igual** que el valor de la Fraccion indicada por parámetro. Generando

un valor **true** que indica que es mayor igual y **false** lo contrario.

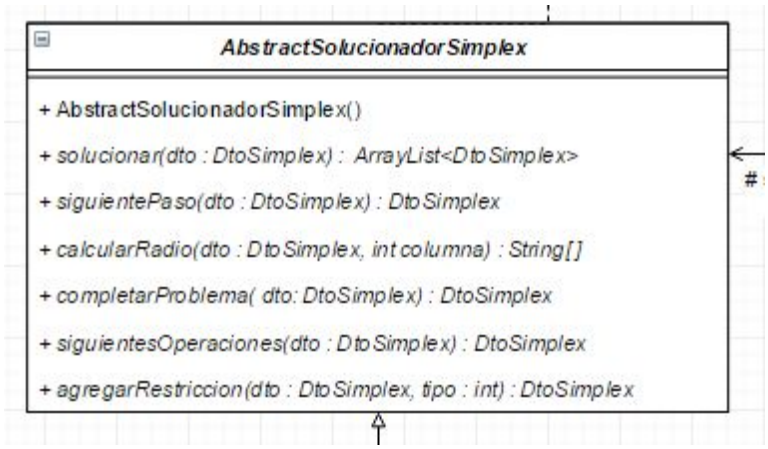
- **mayorQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **mayor** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que es mayor y **false** lo contrario.
- **menorIgualQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **menor igual** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que es menor igual y **false** lo contrario.
- **menorQue(fraccion2 : AbstractFraccion):** compara si el valor numérico de la fracción donde se ejecuta este método es **menor** que el valor de la Fraccion indicada por parámetro. Generando un valor **true** que indica que es menor y **false** lo contrario.
- **hacerNegativa():** Convierte una instancia de Fraccion a su inverso de signo, negando los valores de numerador y denominador.
- **obtenerMayorDivisorComun(numero1 : double, numero2 : double) :** Obtiene el divisor mayor existente entre los números enteros indicados por medio de parámetro, utiliza el algoritmo de euclides.
- **validarSignos():** valida los signos del numerador y denominador dejando un signo negativo si fuese necesario en el numerador.

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



- **Atributos:** esta clase no posee atributos definidos, solamente hereda los atributos de su clase padre AbstractFraccion explicada anteriormente.
- **Constructores:**
  - **Fraccion():** invoca al constructor de su clase padre, creando una nueva instancia de Fraccion con numerador igual a 0 y denominador igual a 1.
  - **Fraccion(valor : double):** invoca al constructor de su clase padre AbstractFraccion(valor:Double.)
  - **Fraccion(numerador : int, denominador: int):** invoca al constructor de su clase padre AbstractFraccion(numerador : int, denominador: int).
- Métodos:
  - **esCero(fraccion : AbstractFraccion) :** analiza si el valor numérico de la fracción indicada por parámetro representa al valor cero.

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



- **Atributos:** no posee atributos.

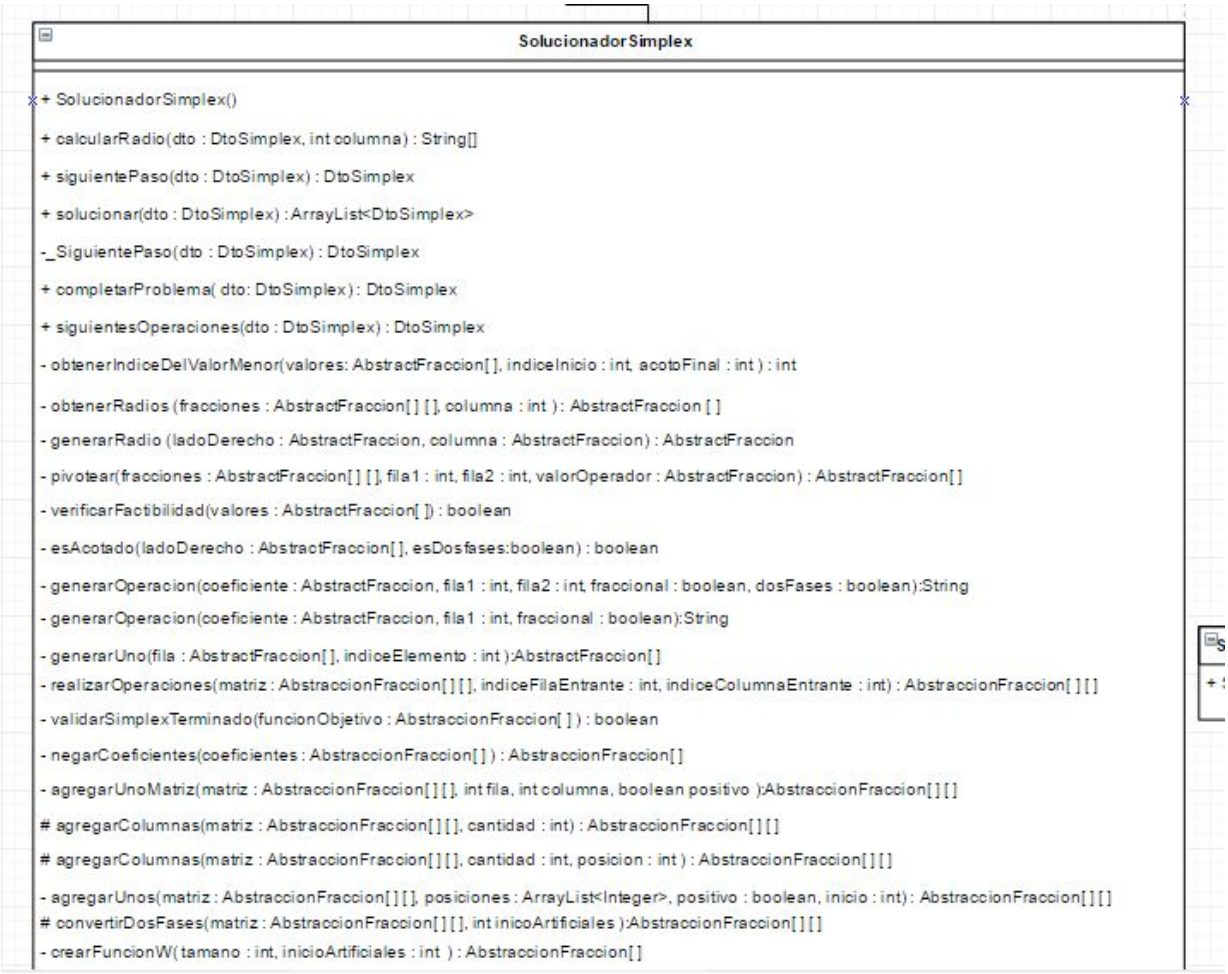
- **Constructores:**
  - **AbstractSolucionadorSimplex():** crea una instancia vacía de la clase.
- **Métodos:**
  - **solucionar(dto:DtoSimplex):** resuelve un prolema de programación lineal, este método es implementado por sus clases concretas y puede actuar de distintas maneras, al final devuelve un arreglo de DtoSimplex que representan el estado del problema en cada iteración.
  - **siguientePaso(dto:DtoSimplex):** genera la siguiente iteración de un problema de programación lineal, este método es implementado por sus clases concretas y puede actuar de distintas maneras, devuelve un DtoSimplex con los datos del problema en la siguiente iteración.
  - **calcularRadios(**dto:DtoSimplex, columna : int**):** genera los radios según la columna indicada por medio del paramétro. Este método es implementado por sus clases concretas y puede actuar de distintas maneras, sino lo define devuelve el valor generado por el método definido por su clase padre. Devuelve un **String[ ]** de los radios según el formato indicado dentro del DtoSimplex.
  - **completarProblema(dto:DtoSimplex):** Agrega las variables artificiales y de holgura a la matriz de coeficientes (AbstractFraccion) contenida dentro del DtoSimplex, además cambia el signo del "lado derecho" y los coeficientes de una fila si el valor del

"lado derecho" es negativo. Este método es implementado por sus clases concretas y puede actuar de distintas maneras, sino lo define devuelve el valor generado por el método definido por su clase padre. Devuelve un DtoSimplex con la matriz y variables completadas.

- **siguientesOperaciones(dto : DtoSimplex):** genera un string almacenado dentro del objeto de transferencia de datos, que indica cuales son las siguientes operaciones filas a realizar. Este método es implementado por sus clases concretas y puede actuar de distintas maneras, sino lo define devuelve el valor generado por el método definido por su clase padre. El String generado se almacena dentro del DtoSimplex para retornar.
- **agregarRestriccion(dto:DtoSimplex, tipo:int):** Agrega una restricción al problema de programación lineal ingresado por parámetro. Esta restricción puede ser mayorIgual(>=), menorIgual(<=) o igual(=), este tipo de restricción se indicará por medio del número entero y el valor de cada restricción se encuentra almacenado por defecto dentro del archivo sym.java. Este método es implementado por sus clases concretas y puede actuar de distintas maneras, sino lo define devuelve el valor generado por el método definido por su clase padre.

- **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[ ], esDosfases:boolean) : 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[ ] [ ], cantidad : int) : AbstraccionFraccion[ ] [ ]
agregarColumnas(matriz : AbstraccionFraccion[ ] [ ], cantidad : int, posicion : int) : AbstraccionFraccion[ ] [ ]
- agregarUnos(matriz : AbstraccionFraccion[ ] [ ], posiciones : ArrayList<Integer>, positivo : boolean, inicio : int) : AbstraccionFraccion[ ] [ ]
convertirDosFases(matriz : AbstraccionFraccion[ ] [ ], int inicoArtificiales):AbstraccionFraccion[ ] [ ]
- crearFuncionW(tamano : int, inicioArtificiales : int) : AbstraccionFraccion[ ]

- **Atributos:** La implementación de esta clase no necesitó atributos.
- **Constructores**:
  - **SolucionadorSimplex():** instancia un objeto de la clase SolucionadorSimplex.
- **Métodos**:
  - **Solucionar(dto:DtoSimplex):** soluciona un problema de programación lineal y obtiene de inmediato un arreglo de DtoSimplex con el estado del problema en cada iteración. Dicho resultado se obtiene por medio de ejecutar el método.
  - **_SiguientePaso** y obtener automáticamente la iteración las iteraciones. Previamente antes de ejecutar este método el DtoSimplex que representa al problema enviado por parámetro debe de estar completado que se obtiene ejecutando el método **completaProblema**.
  - **siguientePaso(dto : DtoSimplex):** genera la siguiente iteración del problema de programación lineal, invoca al método auxiliar **_SiguientePaso**.

- **_SiguientePaso(dto : DtoSimplex):** método auxiliar creado para eliminar dependencias cíclicas de llamado de métodos que se encuentra en las clases hijas que heredan de SolucionadorSimplex. Este método se encarga de validar si el problema se encuentra en la primera fase o en la segunda del método Simplex, para ejecutar el método correspondiente. Retorna un DtoSimplex con los datos del problema en la siguiente iteración.
- **calcularRadios(dto : DtoSimplex, columna : int):** genera un arreglo de tipo String, donde se calcula el radio de la columna número indicada por parámetro, por el RHS que es la última columna de la matriz que representa al problema almacenada dentro del DtoSimplex. Dichos Strings se obtienen por ejecutar el método **toString(fraccional:boolean),** de los elementos tipo AbstractFraccion de la matriz. El dato tipo fraccional siempre va a ser **false por petición del cliente.**
- **completarProblema(dto : DtoSimplex):** se encarga de agregar las columnas necesarias según el tipo de restricción indicado por parámetro dentro del DtoSimplex. Además agrega el nombre que le corresponde a una variable ya sea en la columna o si es necesario en las filas. Agrega la característica al DtoSimplex de ser o no resuelto por medio del método de dos fases y agrega la fila que representa la función -w. Inicializa la coordenada para comenzar a iterar sobre el problema. Devuelve un DtoSimplex con los todos los datos generados agregados listos para comenzar a solucionar el problema.
- **siguientesOperaciones(dto:DtoSimplex):** crea un arreglo de elementos tipo String, que representan a las operaciones fila que se realizarán según la casilla indicada por la coordenada que se encuentra como atributo dentro del DtoSimplex. Dicho arreglo se almacena dentro del DtoSimplex y es retornado.
- **generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fila2 : int, fraccional : boolean, dosFases : boolean):** genera un String que representa una operación fila cuando se

involucran más de una fila. El coeficiente poseerá el formato indicado por parámetro y si el problema se encuentra en la etapa de dos fases la enumeración(num) de las filas comenzará desde la 0' continuando luego con el conteo normal. El formato del String resultante sigue el patrón siguiente:

Coeficiente * F num(fila1) + F num(fila2) -> F num(fila2)

- **generarOperacion(coeficiente : AbstractFraccion, fila1 : int, fraccional : boolean):** genera un String que representa una operación fila cuando se involucran unicamente una fila. El coeficiente poseerá el formato indicado por parámetro, la numeración es normal no es necesario validar si se encuentra en el método de dos fases. El formato del String resultante sigue el patrón siguiente:

Coeficiente * F num(fila1) -> F num(fila1)

- **obtenerIndiceDelValorMenor(valores: AbstractFraccion[ ], indiceInicio : int, acotoFinal : int):** Recorre un arreglo de elementos tipo AbstractFraccion comenzando desde el índice de inicio y culminando en el índice generado por cantidad de elementos menos el acotoFinal. El fin de recorrer la lista se basa en encontrar el elemento menor de la lista y retornar el índice de donde se encuentra.
- **obtenerIndiceDeSiguienteVariableSaliente(valores, AbstractFraccion[], indiceInicio : int, acotoFinal : int, dtoProblema : DtoSimplex):** Obtiene el índice del valor menor que se encuentre dentro de la lista de fracciones que representa al lado derecho, si el algoritmo se encuentra en el método de dos fases se le da prioridad a las variables artificiales.
- **obtenerRadios (fracciones : AbstractFraccion[ ] [ ], columna : int):** realiza la división entre una columna indica por parámetro y el "*lado derecho*" que se encuentra en la última posición de la matriz indicada por parámetro. Dicha división se realiza dividiendo las posiciones correspondientes entre columnas.

- **generarRadio (ladoDerecho : AbstractFraccion, columna : AbstractFraccion):** divide el elemento del *"lado derecho"* entre el de la columna. Dicha división si es menor a 0 o el denominador es 0 retorna infinito. En nuestro caso el infinito es representado por medio de una Fraccion con denominador igual a Double.maxValue.
- **pivotear(fracciones : AbstractFraccion[ ] [ ], fila1 : int, fila2 : int, valorOperador : AbstractFraccion) :** realiza la acción de pivotear del método simplex. Esta operación se ejecuta entre dos filas de la matriz y ejecutando la siguiente operación entre las posiciones correspondientes: (valorOperador ***** elemetofila1) **+** elementofila2
- **verificarFactibilidad(valores : AbstractFraccion[ ]) :** Verifica que si los valores de la fila indicada son todos mayores o igual que 0, el elemento que se encuentra en la última posición debe ser igual a 0. En caso de no cumplirse estas características retorna **false** indicando que no es factible y del contrario **true**.
- **esAcotado(ladoDerecho : AbstractFraccion[ ], esDosFases:boolean) :** Verifica si los elementos generados como radios de el *"lado derecho",* brindan la característica de que el problema está acotado o no. Este valor se obtiene comparando los elementos del radio que pertenezcan a las restricciones, por ellos se le indica si el problema se encuentra en la etapa de dos fases para omitir la primera o las dos primeras filas. Para obtener si se encuentra acotado se compara cada radio si es igual a nuestro **infinito** si existe al menos uno diferente retorna **true** y si no **false** que representa si está acotado.
- **generarUno(fila : AbstractFraccion[ ], indiceElemento : int):** crea un un uno en la posición indicada dentro del arreglo de fracciones. Para generar dicho uno se divide toda la fila de fracciones por el elemento indicado en la posición.
- **realizarOperaciones(matriz : AbstraccionFraccion[ ] [ ], indiceFilaEntrante : int, indiceColumnaEntrante : int) :** realiza todas

las operaciones fila necesarias para que la variable que se encuentra en la fila y columna indicada sea variable básica. Este resultado se obtiene por medio de pivotear todas las filas por la fila entrante utilizando como coeficiente el inverso de signo del elemento en las otras filas en la columna indicada.

- **validarSimplexTerminado(funcionObjetivo : AbstraccionFraccion[ ]) :** válida si los coeficiente de la función objetivo de un problema de programación lineal se encuentra en su estado óptimo. Dicho estado se encuentra comparando los elementos del arreglo de fracciones que todos sean mayor que 0.
- **negarCoeficientes(coeficientes : AbstraccionFraccion[ ]) :** invierte el signo de las fracciones del arreglo indicado por parámetro.
- **agregarUnoMatriz(matriz : AbstraccionFraccion[ ] [ ], int fila, int columna, boolean positivo):** agrega una Fraccion con el valor de 1,en la fila y columna indicada. Dicho uno puede ser negativo o positivo dependiendo del parámetro indicado. Retorna la matriz con el uno indicado.
- **agregarColumnas(matriz : AbstraccionFraccion[ ] [ ], int cantidad) :** agrega la cantidad de columnas indicadas a la matriz. Dicha operación mantiene la última columna en su posición debido a que el algoritmo la utiliza como *"lado derecho"* y es necesario que se mantenga para realizar distintas operaciones del método de dos fases. Retorna la matriz con las columnas agregadas.
- **agregarColumnas(matriz : AbstraccionFraccion[ ] [ ], cantidad : int, posicion : int) :** agrega la cantidad de columnas indicadas a la matriz comienza la adicción en la posición indicada. Dicha operación mantiene la última columna en su posición debido a que el algoritmo la utiliza como *"lado derecho"* y es necesario que se mantenga para realizar distintas operaciones del método de dos fases. Retorna la matriz con las columnas agregadas.
- **agregarUnos(matriz : AbstraccionFraccion[ ] [ ], posiciones : ArrayList<Integer>, positivo :**

boolean, inicio : int) : agrega unos utilizando la función agregarUno, en las filas que se encuentran en el arreglo de posiciones y columnas secuencialmente iniciando desde el parámetro inicio, dichos unos se indicará el signo por medio del parámetro booleano. Retorna una matriz con los unos agregados.

- **convertirDosFases(matriz : AbstraccionFraccion[ ] [ ], int inicioArtificiales):** convierte la estructura de la matriz del problema de programación lineal en un problema de dos fases. Agrega la fila que representa a la función w conjunto a sus variables artificiales. Retorna la matriz convertida a dos fases.
- **crearFuncionW(tamano : int, inicioArtificiales : int) :** crea un arreglo de elementos tipo Fraccion de un tamaño indicado que representa a la función w. Dicho arreglo se le agregan los unos de las variables artificiales iniciando desde el índice indicado.
- **generarCoeficiente(matriz:AbstractFraccion[][], indiceFila : int, indiceColumna : int):** crea un coeficiente tomando el elemento que se encuentra en la fila y columna indicado. Dicho elemento se le invierte el signo y es retornado como coeficiente.
- **siguientePivoteo(dto : DtoSimplex):** crea el siguiente punto donde se realizará un pivoteo, busca la columna que contenga en la función objetivo el valor menor y el radio de la restricción con valor menor. Con los índices encontrados genera un Punto que representará las siguientes operaciones y variable entrante.
- **agregarNombreVariables(nombres : String[], cantidad : int, nuevoNombre : String):** agrega una cantidad de nombres a un arreglo de elementos tipo String. Los nuevos nombres son generados por medio del siguiente patrón: **nuevoNombre+indiceColumna**. Retorna un nuevo arreglo con los nombres agregados.
- **agregarNombresFila(nombres : String[], Indicesfila : ArrayList<Integer>, nombreFila : String, indiceInicio : int) :** crea los nombres de las

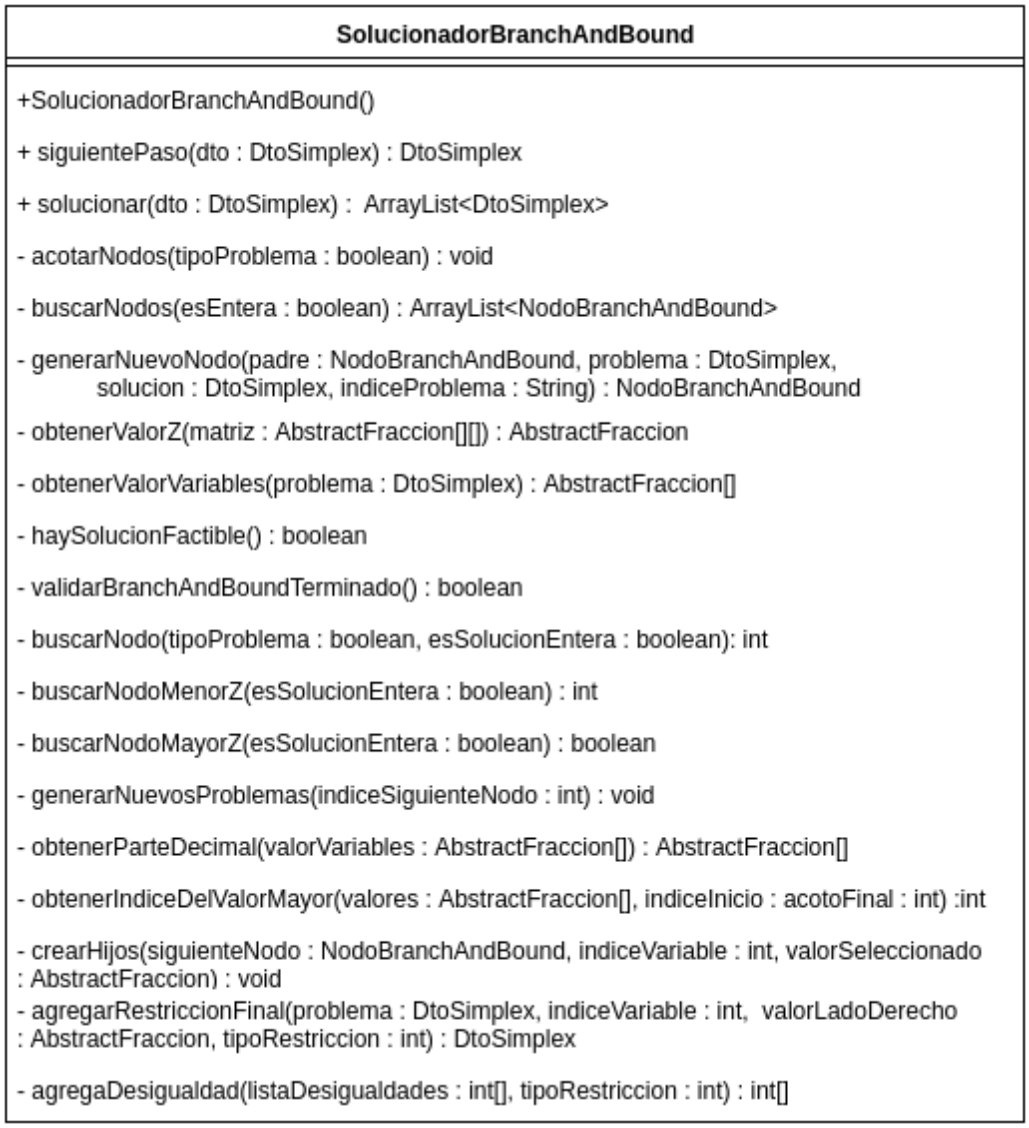
filas que representan a las variables seleccionadas. Dichos nombres se crean por medio del sufijo indicado por parámetro y el índice de incio. A cada fila se le agrega el nombre correspondientemente se encuentre en el arreglo de índices.

- **agregarNombreW(nombres : String []):** agrega al inicio del nombre de las filas, el String “-w” que representa a la función objetivo de un problema de dos fases.
- **crearNombreFila(tamano : int, nombreCabeza :String):** Inicializa el arreglo de elementos tipo String que representan a los nombres de los nombres de fila, se le asigna el nombre “z” y “-z” dependiendo del tipo de problema.
- **siguientePasoSimplex(dto : DtoSimplex) :** realiza las operaciones de la iteración actual del problema de programación lineal dentro del DtoSimplex. Para ejecutar un siguientePaso es necesario que el problema se encuentre completado. Realiza las siguientes operaciones según el punto dentro del DtoSimplex. Valida si el problema fue terminado y se encuentra en estado óptimo. Antes de iterar se valida si el problema esta acotado. El método cerciora si el problema es de dos fases para indicar cuando termina una etapa. Devuelve un DtoSimplex con los datos de la iteración.
- **siguientePasoDosFases(dto : DtoSimplex) :** se encarga de ejecutar las acciones correspondientes cuando el problema se encuentra en la sección de dos fases. Si se encuentra eliminando las variables artificiales del método de dos fases ejecuta una acción distinta a cuando se continúa con el flujo normal de terminar la primera fase. También valida si el problema es factible que es la sección donde sucede. Devuelve un DtoSimplex con los datos de la iteración.
- **eliminarArtificiales(dto : DtoSimplex):** método invocado al inicio de la primera fase del simplex de fases, se encarga de eliminar las variables artificiales que poseen un uno en la función objetivo. Genera un punto en la fila donde se encuentra seleccionada la variable artificial de la siguiente

iteración.Válida cuando se eliminaron todos los unos que representan a las variables artificiales para continuar con el método normal del Simplex. Devuelve un DtoSimplex con los datos después de la iteración.

- **buscarIndice(nombres : String[], nombre : String):** busca un String dentro de un arreglo de elementos del mismo tipo y retorna el índice donde se encuentra. Si no lo encuentra retorna -1.
- **eliminarFilaW(dto : DtoSimplex):** Elimina la primera fila de la matriz que se encuentra dentro del DtoSimplex. Esta fila representa la función w del método de dos fases y se utiliza cuando termina la primera etapa. Además elimina del nombre de las filas el -w.
- **eliminarColumnasArtificiales(dto : DtoSimplex):** elimina de la matriz que se encuentra dentro del DtoSimplex, las columnas que representan a las variables artificiales. Este método se utiliza cuando termina la primera etapa del Simplex de dos fases y retorna la matriz almacena dentro del DtoSimplex.
- **eliminarNombreW(nombresFila : String[]):** Toma el arreglo de elementos tipo String y elimina el primer elemento que representa a la variable -w. Este método se utiliza cuando termina la primera etapa del Simplex de dos fases y retorna un nuevo arreglo sin el primer elemento.
- **eliminarNombresColumnas(nombresColumnas : String[], indiceInicioArtificiales : int):** elimina del arreglo de elementos tipo String que representa al nombre de las columnas, los elementos que representan las columnas de las variables artificiales que inician desde el índice indicado. Este método se utiliza cuando termina la primera etapa del Simplex de dos fases y retorna un nuevo arreglo sin el nombre de las artificiales.
- **siguientesOperacionesInicioDosfases(indice :int):** genera un String que representa una operación de fila que se realiza al inicio del método de dos fases, cuando se eliminan las variables artificiales. El String retornado posee el siguiente formato: "**-** F inidice + F0' -> F0' "

        - **obtenerSolucion(dto : DtoSimplex):** Retorna un dato tipo String con el valor representativo de la solución del problema de programación lineal. Solo incluye las variables de la función objetivo original.
        - **agregarFila(matriz : AbstractFraccion[][]):** agrega una fila de elementos tipo AbstractFraccion a la matriz indicada por parámetro. Dicha fila solo contiene valores de 0.
        - **agregarRestriccion(dto : DtoSimplex, tipo : int):** agrega una nueva restricción a la matriz de elementos tipo AbstractFraccion. El tipo int dependerá de los definidos dentro de las constantes en el archivo sym.java.
        - **agregarMenorIgual(dto : dtoSimplex):** agrega una restricción tipo menor o igual (<=) a la matriz de elementos tipo AbstractFraccion. Actualiza la cantidad de columnas y las variables ligadas al número de variables de holgura y artificiales dentro del DtoSimplex y lo retorna.
        - **agregarMayorIgual(dto : dtoSimplex):** agrega una restricción tipo menor o igual (>=) a la matriz de elementos tipo AbstractFraccion. Actualiza la cantidad de columnas y las variables ligadas al número de variables de holgura y artificiales dentro del DtoSimplex y lo retorna.
        - **agregarIgual(dto : dtoSimplex):** agrega una restricción tipo menor o igual (=) a la matriz de elementos tipo AbstractFraccion. Actualiza la cantidad de columnas y las variables ligadas al número de variables de holgura y artificiales dentro del DtoSimplex y lo retorna.
- **SolucionadorBranchAndBound:**
    - **Descripción:** Posee la implementación de los métodos de AbstractSolucionadorSimplex y SolucionadorSimplex. Maneja el caso de uso de solucionar un problema mediante el algoritmo de Branch and Bound. Debido a que el algoritmo utiliza un arbol de soluciones, se implementa una clase privada Nodo por este motivo.
    - **UML:**



- **Atributos:**
  - **arbol : NodoBranchAndBound:** referencia al nodo raíz del árbol generado por el algoritmo de branch and bound.
  - **hojas : ArrayList<NodoBranchAndBound>:** arreglo dinámico de nodos que representan a las hojas del árbol. Se creó esta atributo para evitar búsquedas de hojas por cada iteración en el árbol.
- **Constructores:**
  - **SolucionadorBranchAndBound():** instancia un objeto que soluciona problemas de programación lineal por medio del algoritmo de Branch and Bound. Inicializa el arreglo de hojas miembro de la clase.
- **Métodos:**
  - **solucionar(dto:DtoSimplex):** itera las veces posible sobre el problema de programación lineal

ingresado hasta llegar a una solución o encontrar que el problema no es factible. El arreglo de elementos tipo DtoSimplex contiene el estado del árbol como solución y un mensaje con lo sucedido en el árbol.

- **siguientePaso(dto:DtoSimplex):** itera un nodo hoja para generar dos nuevas soluciones. También se encarga de verificar si el árbol se encuentra en la primera iteración, en este caso solamente genera un nodo hoja. Verifica si el problema se soluciono con valores enteros y si existe una solución factible. Cada iteración llama al método **solucionar** de su clase padre **SolucionadorSimplex,** con la cual se crea una nueva iteración del problema.
- **acotarNodos(tipoProblema : boolean):** ejecuta una búsqueda de nodos tipo NodoBranchAndBound dentro de las hojas del árbol, donde no se encuentren acotados su solución y sean no enteras. Esto con el fin de comparar con las soluciones enteras existentes y verificar si se puede acotar un nodo. Un nodo se acota dependiendo del tipo de problema en que se encuentre ya sea de minimización o maximización, si es de **maximización** se acota si existe una solución entera con un valor z mayor o igual y si fuese **minimización** sería con un valor z menor o igual.
- **buscarNodos(esEntera : boolean) :** realiza una búsqueda dentro del arreglo de hojas del árbol del problema que se ha generado. El fin de esta búsqueda es encontrar nodos con solución factible y acotada, y además si su solución es entera o no dependiendo del valor del parámetro esEntera, si es **true** la solución es entera y del contrario es **false**.
- **generarNuevoNodo(padre : NodoBranchAndBound, problema : DtoSimplex, solucion : DtoSimplex, indiceProblema : String) :** genera una nueva hoja tipo NodoBranchAndBound de un problema anterior, al nodo generado se le asigna el valor z, el valor de las variables objetivo del problema, el índice del problema y además se verifica si el nuevo nodo posee una solución factible.

- **obtenerValorZ(matriz : AbstractFraccion[][]):** retorna el valor solución del problema actual, recibe la matriz de fracciones y retorna el valor de la función objetivo, el cual se encuentra en la última posición de la primera fila.
- **obtenerValorVariables(problema : DtoSimplex):** obtiene la cantidad de variables objetivo del problema y busca acorde la fila donde se encuentre seleccionada y su valor derecho. Dichos valor son devueltos como el valor de la variable y según como se encuentre ordenadas las filas se le asigna un valor en el arreglo de Fraccion que se va a retornar, si una variable no se encuentra seleccionada su valor es dado por 0.
- **haySolucionFactible() :** recorre el arreglo de hojas del árbol verificando si existe una solución factible. Si hay al menos una hoja con solución factible retorna **true** y si no **false**.
- **validarBranchAndBoundTerminado():** recorre el arreglo de hojas del árbol verificando si existe aún un nodo con una solución no entera factible y que no se encuentre acotada.Si encuentra un nodo con las características anteriores retorna **false** y de los contrario **true**.
- **buscarNodo(tipoProblema : boolean, esSolucionEntera : boolean):** Busca dentro del arreglo de hojas, el índice del siguiente nodo con solución de z mejor entera o no. Dicho nodo dependerá del tipo de problema que se indique, se retorna el nodo con mayor valor z si el problema es de maximización y el menor si el problema es de minimización.
- **buscarNodoMenorZ(esSolucionEntera : boolean):** busca dentro del arreglo de hojas del árbol generado, el índice de la hoja con el menor valor z. Dicho valor no debe de estar acotado y ser factible. Además el valor z podría ser entera o no dependiendo del parámetro esEntera donde **true** indica que sea entera de lo contrario **false**.
- **buscarNodoMayorZ(esSolucionEntera : boolean):** busca dentro del arreglo de hojas del árbol generado, el índice de la hoja con el mayor

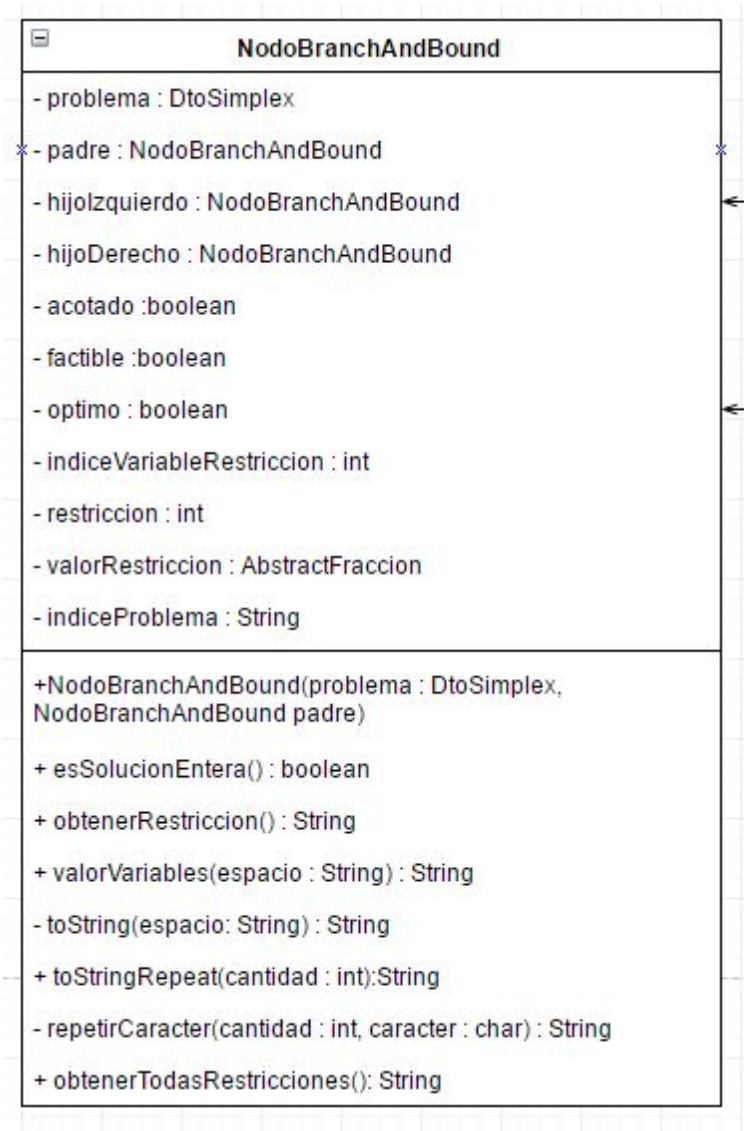
valor z. Dicho valor no debe de estar acotado y ser factible. Además el valor z podría ser entera o no dependiendo del parámetro esEntera donde **true** indica que sea entera de lo contrario **false**.

- **generarNuevosProblemas(indiceSiguienteNodo : int):** se encarga de generar los dos nuevos nodo hijos de un nodo seleccionado, se indica el nodo por medio del índice pasado por parámetro. Además se encarga de obtener la variable con la cual se realizarán las nuevas restricciones de las ramas del árbol.
- **obtenerParteDecimal(valorVariables : AbstractFraccion[]) :** separa la parte decimal de los valores numéricos reales que representan las fracciones que se encuentran en el arreglo indicado. Devuelve un arreglo donde se encuentran únicamente la parte decimal de cada valor respectivamente.
- **obtenerIndiceDelValorMayor(valores : AbstractFraccion[], indiceInicio : acotoFinal : int):** Recorre un arreglo de elementos tipo AbstractFraccion comenzando desde el índice de inicio y culminando en el índice generado por cantidad de elementos menos el acotoFinal. El fin de recorrer la lista se basa en encontrar el elemento mayor de la lista y retornar el índice de donde se encuentra.
- **crearHijos(siguienteNodo : NodoBranchAndBound, indiceVariable : int, valorSeleccionado : AbstractFraccion):** agrega las nueva restricciones que implica la iteración en el problema generado hasta el nodo actual. Con ello se crea dos nodos nuevos donde se resuelve el problema creado y se agrega el valor del z, el valor de las variables y verifica si el problema es factible o no. Elimina el nodo padre y agrega los dos nuevos nodos generados al arreglo de hojas del árbol.
- **agregarRestriccionFinal(problema : DtoSimplex, indiceVariable : int, valorLadoDerecho : AbstractFraccion, tipoRestriccion : int):** agrega una nueva fila a la matriz contenida dentro del DtoSimplex. Donde se crea una nueva restricción

dependiendo del tipo indicado por parámetro, dichos tipos se encuentran en el archivo sym.java. Agrega un uno en la posición de la variable seleccionada y actualiza la lista de desigualdades dentro del DtoSimplex.

- **agregaDesigualdad(listaDesigualdades : int[], tipoRestriccion : int):** actualiza la lista de desigualdades que representan a las restricciones de un problema de programación lineal. Agrega al final de la lista el valor representativo entero de la nueva restricción.

- **NodoBranchAndBound:**
  - **Descripción:** implementación de un árbol binario para la ejecución del algoritmo Branch and Bound. Almacena las características de de un problema con todas las restricciones agregadas hasta llegar al estado actual. Esta clase es privada y es únicamente utilizada por el solucionadorBranchAndBound.java.
  - **UML:**



- **Atributos:**
  - **problema:DtoSimplex:** contiene todas las características del problema de programación lineal con las restricciones agregadas recorriendo el árbol que genera el algoritmo de Branch And Bound.
  - **padre:NodoBranchAndBound:** referencia al nodo padre del árbol donde se encuentra el nodo. Si este es nulo significa que es el nodo raíz del árbol.
  - **hijolzquierdo:NodoBranchAndBound:** referencia al hijo izquierdo del nodo actual. Si es nulo significa que es un nodo hoja y no posee hijo derecho.
  - **hijoDerecho:NodoBranchAndBound:** referencia al hijo derecho del nodo actual. Si es nulo significa que es un nodo hoja y no posee hijo izquierdo.
  - **acotado:boolean:** valor binario que indica si el problema o nodo actual fue acotado porque existe

otra mejor solución. Si es **true** significa que está acotado, **false** de lo contrario.

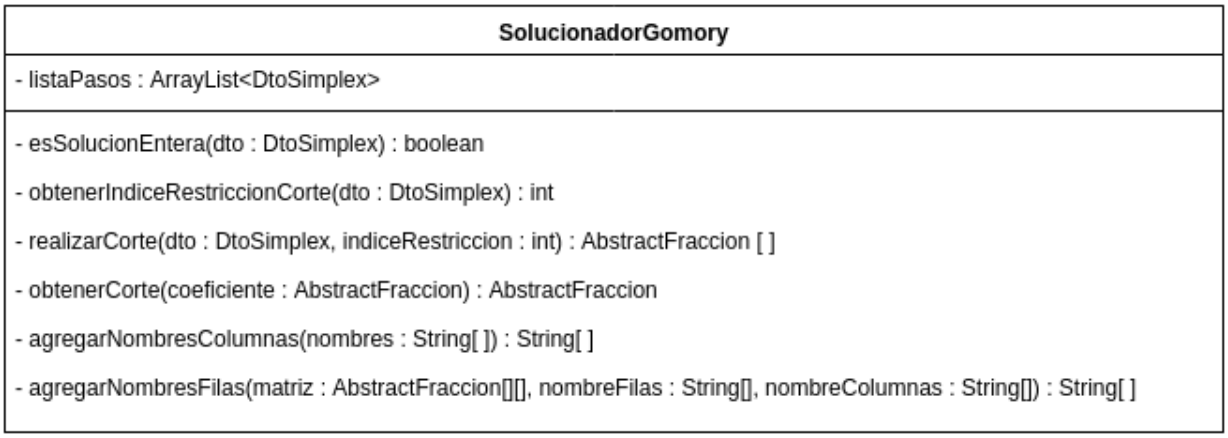
- **factible:boolean:** indica si el nodo actual con la restricción agregada posee una solución óptima factible. Si es **true** significa que el problema es factible, **false** de lo contrario.
- **optimo : boolean:** indica que si el nodo actual posee la solución óptima generada por el algoritmo de Branch And Bound. Si es **true** significa que es un óptimo, **false** de lo contrario.
- **indiceVariableRestriccion : int:** contiene el índice de la variable elegida para crear una nueva restricción, dicho valor indexa el arreglo de nombre de columnas almacenada entre el DtoSimplex.
- **restriccion : int :** valor entero que indica el tipo de restricción agregada por el algoritmo de branch and bound. Dicho entero que representa a las restricciones de encuentran en el archivo sym.java.
- **valorRestriccion : AbstractFraccion:** representa al lado derecho de la restricción agregada. El valor es entero pero se encapsula en una instancia de AbstractFraccion.
- **indiceProblema : String:** valor String que indexa los problemas de programación lineal dentro del árbol representado en String.

- **Constructores:**
  - **NodoBranchAndBound(problema : DtoSimplex, NodoBranchAndBound padre):** inicializa una instancia de NodoBranchAndBound donde se asigna el problema y la referencia al nodo padre.
- **Métodos:**
  - **esSolucionEntera() :** verifica que el valor de todas las variables que forman parte de la función objetivo poseen valores enteros. Si se cumple esta condición retorna **true** de lo contrario **false**.
  - **obtenerRestriccion():** genera un String con los valores de la variable elegida para la restricción, el tipo de restricción agregada y el valor de la nueva restricción en el *"lado derecho"*. El resultado posee el siguiente formato: **variable** (**desigualdad**) **valor**.
  - **valorVariables(espacio : String):** genera un String con el valor de las variables del problema contenido

dentro del DtoSimplex. El string contiene el valor del resultado de la función objetivo y de las variables que la conforman, por cada salto de línea posee un espacio indicado por parámetro.

- **toString(espacio: String):** Genera un String de los datos del nodo, indica si es acotado, factible o muestra todo el contenido con un espacio de tabulación, dicho contenido incluye el valor de las variables ejecutando el método **valorVariables(espacio : String)**.
- **toStringRepeat(cantidad : int):** Genera el String del arbol que representa el nodo actual. Posee la estructura similar a la de un file system, con los nodos como elementos. Estrucutra: Problema 1 información del nodo Problema 1.1 Información del nodo, la información del nodo se genera por medio de la función **toString(espacio: String),** donde se crea la cantidad de espacio de tabulación según el parámetro llamado **cantidad**.
- **repetirCaracter(cantidad : int, caracter : char) :** repite la cantidad de veces dentro de un String el caracter que se indique por parámetro.
- **obtenerTodasRestricciones():** recorre recursivamente el arbol donde se encuentra el nodo donde se ejecute este método, tomando las restricciones agregadas y almacenandolas en un solo String. El nodo raíz no posee restricción y se le agrega el dato “No es necesario”.

- **SolucionadorGomory:**
  - **Descripción:** Posee la implementación de los métodos de AbstractSolucionadorSimplex y SolucionadorSimplex. Maneja el caso de uso de solucionar un problema mediante el algoritmo de Cortes de Gomory.
  - **UML:**



- **Atributos:**
  - **listaPasos : ArrayList<DtoSimplex>:** arreglo de DtoSimplex que almacena el estado del problema de programación lineal en cada iteración del algoritmo de cortes de Gomory.
- **Constructores:**
  - **solucionar(dto:DtoSimplex):** itera las veces posible sobre el problema de programación lineal ingresado hasta llegar a una solución o encontrar que el problema no es factible. El arreglo de elementos tipo DtoSimplex contiene el estado del problema tras ejecutar el algoritmo de Gomory.
  - **siguientePaso(dto:DtoSimplex):** obtiene la siguiente iteración del problema tras ejecutar o añadir restricciones dadas por el algoritmo de Gomory.
  - **esSolucionEntera() :** verifica que el valor de todas las variables que forman parte de la función objetivo poseen valores enteros. Si se cumple esta condición retorna **true** de lo contrario **false**.
  - **obtenerIndiceRestriccionCorte(dto : DtoSimplex):** obtiene el índice de la variable seleccionada en una fila con la cual se añadirá el siguiente corte de gomory.
  - **realizarCorte(dto : DtoSimplex, indiceRestriccion : int):** genera un nuevo corte con el índice de la fila seleccionada, toma la fila de la matriz y obtiene los valores decimales de los coeficientes para crear la nueva restricción o corte del problema.
  - **obtenerCorte(coeficiente : AbstractFraccion):** obtiene de un coeficiente el valor de la parte decimal para generar un nuevo corte de Gomory.

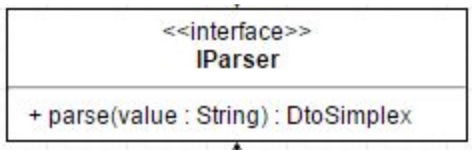
- **agregarNombresColumnas(nombres : String[ ]):** agrega dos nuevos nombres de columna una artificial y otra de holgura. Debido a la nueva restricción es mayor igual.
- **agregarNombresFilas(matriz : AbstractFraccion[][], nombreFilas : String[], nombreColumnas : String[]):** Busca cuáles son las variables básicas de cada fila y pone el arreglo de string respectivo para retornar con la nueva restricción agregada.

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). Para conocer más a fondo las características de las clases generadas visitar la página oficial de CUP incluida en los anexos.

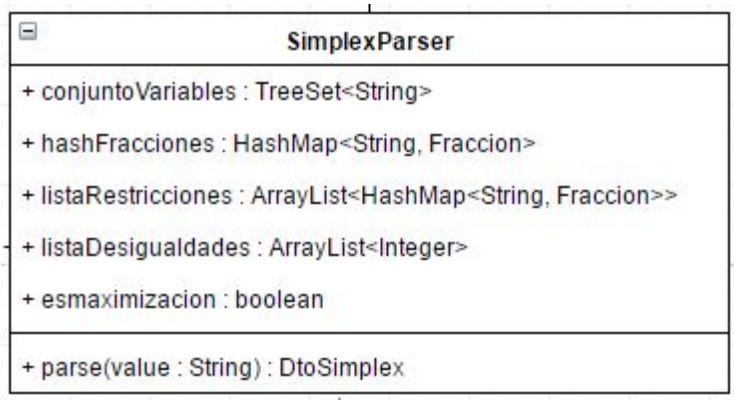
Clases e Interfaces

- **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:**

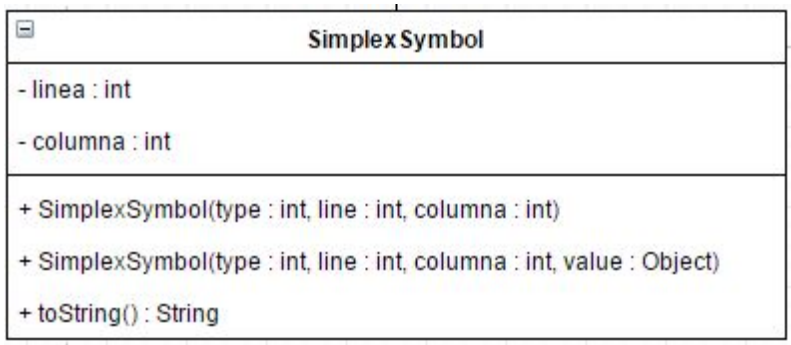


- **SimplexParser:**
  - **Descripción:** Clase autogenerada por CUP. Se encarga de validar e interpretar una cadena de texto que contenga un problema de programación lineal con el formato válido aceptado por el programa.
  - **UML:** El siguiente diagrama UML solamente posee los métodos agregados por 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

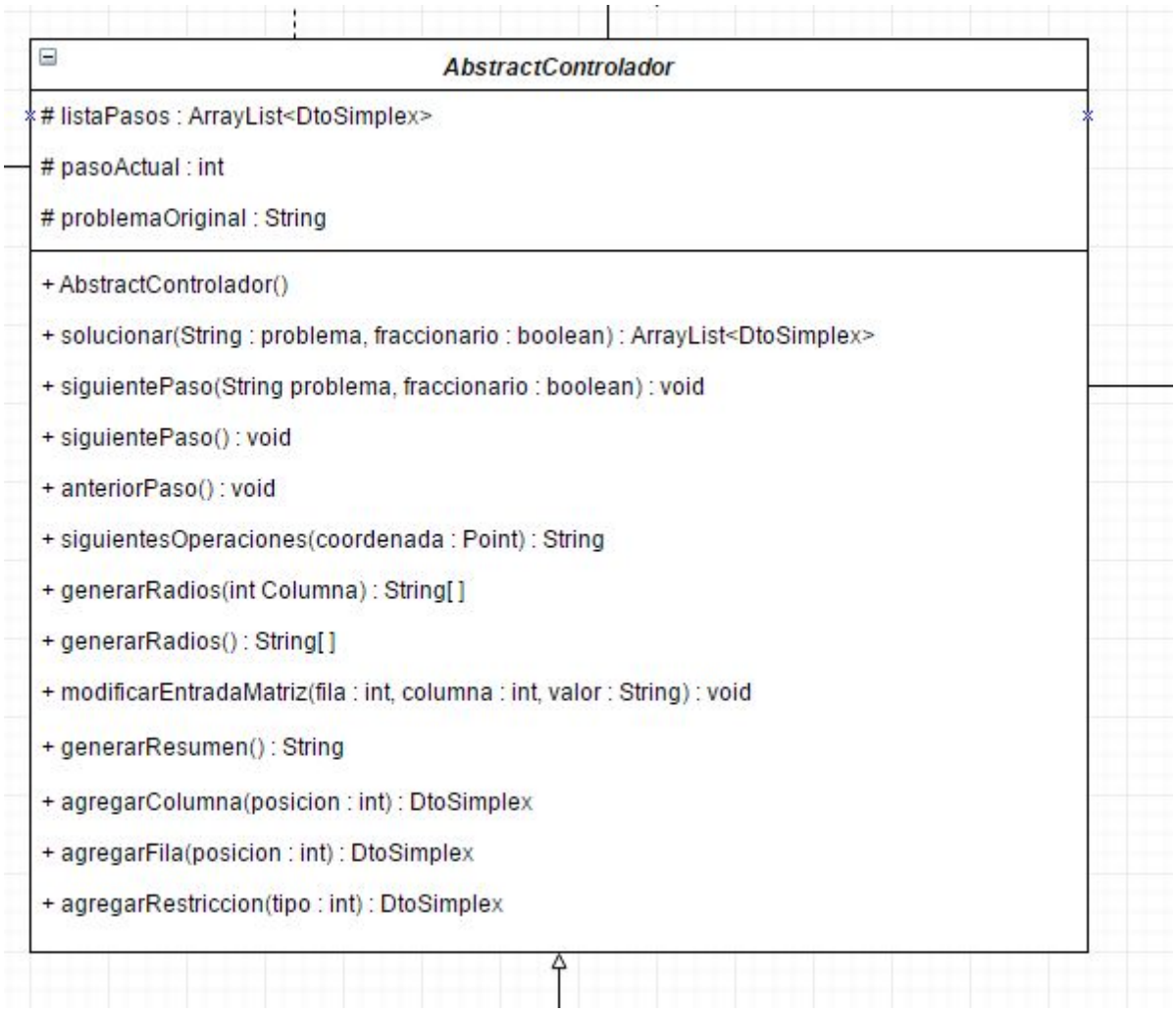
por el scanner. Cada tipo de token tiene asignado un valor entero que está definido en esa clase.
    - **UML:** Clase autogenerada. Solamente posee valores constantes enteros, no es necesario una representación en el diagrama UML.

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:**



- **Atributos:**
  - **solucionador:AbstractSolucionadorSimplex:** provee la solución de los problemas de programación lineal utilizando el método Simplex. Este atributo puede tomar distintas clases concretas gracias al polimorfismo.
  - **parser:IParser:** ejecuta la validación de la cadena de texto que representa al problema de programación lineal.
  - **vista:IVista:** ejecuta las acciones que se mostrarán en la pantalla del usuario. Esta interfaz es aplicada en los formularios creados como parte visual del proyecto.
  - **pasoActual:int:** valor numérico que indica la iteración actual en la que se encuentra el controlador y se está mostrando al usuario.
  - **listaPasos : ArrayList<DtoSimplex>:** arreglo dinámico que contiene las iteraciones que se han

generado sobre el problema de programación. El atributo **pasoActual** indica por cuál de sus elementos se encuentra y no puede ser mayor que la cantidad de elementos de este arreglo.

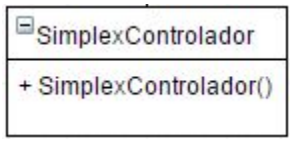
- **problemaOriginal : String:** cadena de texto que representa al problema de programación lineal ingresado por el usuario al inicio de la ejecución.

- **Constructores:**
  - **AbstractControlador():** instancia un objeto tipo AbstractControlador sin inicializar ningún atributo.
- **Métodos:**
  - **solucionar(String : problema, fraccionario : boolean):** ejecuta la validación del problema ingresado por medio del parser. Completa el problema ejecutando el método **completarProblema** del atributo solucionador y con este mismo soluciona el problema por medio del método **solucionar** obteniendo inmediatamente todas las iteraciones intermedias para obtener un resultado sobre el problema ingresado. El resultado se almacena en el atributo listaPasos. Actualiza la vista con los resultados del problema.
  - **siguientePaso(String problema, fraccionario : boolean):** inicializa la ejecución de la solución de un problema por medio iteraciones intermedias. Ejecuta la validación del problema ingresado por medio del parser. Completa el problema ejecutando el método **completarProblema** del atributo solucionador y agrega el problema completado a la lista de pasos. Actualiza la vista con los datos creados.
  - **siguientePaso():** itera sobre el elemento de la lista de pasos donde se encuentra el paso actual. Utiliza el método **siguientePaso** del solucionador. Luego valida si la iteración brinda que el problema es factible, acotado o se logró llegar una solución óptima. Actualiza la vista con los datos de la siguiente operación.
  - **anteriorPaso():** decrementa el contador de pasoActual. Actualiza la vista con los datos del paso anterior.

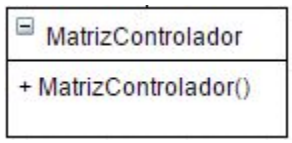
- **siguientesOperaciones(coordenada : Point) :** obtiene un String que representan a las siguientes operaciones sobre el elemento de la listaPasos donde encuentre el contador pasoActual. Dicho cadena de texto se obtiene por medio del método **siguientesOperaciones** del solucionador.
- **generarRadios(int Columna):** genera un arreglo de String con el valor representativo de los radios generados en la columna indicada, cada valor es correspondientemente por cada fila. Si se encuentre el infinito agrega un String representativo ya sea “oo” o “-oo”.
- **generarRadios() :** genera un arreglo de String con el valor representativo de los radios generados en la columna indicada dentro del punto que se encuentra en el paso actual de las iteraciones, cada valor es correspondientemente por cada fila. Si se encuentre el infinito agrega un String representativo ya sea “oo” o “-oo”.
- **modificarEntradaMatriz(fila : int, columna : int, valor : String) :** Cambia el valor de una entrada en la matriz de coeficientes actual, por un nuevo valor en la posición fila y columna indicado dentro de la iteración del paso actual.
- **generarResumen():** recorre el arreglo de las iteraciones (**listaPasos**), ejecutando el método toString de cada DtoSimplex.
- **agregarColumna(posicion : int) :** modifica la matriz que se encuentra dentro del DtoSimplex de la iteración actual. Agrega una columna en la posición indicada ejecutando el método **agregarColumna** del solucionador.
- **agregarFila(posicion : int) :** modifica la matriz que se encuentra dentro del DtoSimplex de la iteración actual. Agrega una fila en la posición indicada ejecutando el método agregarFila del solucionador.
- **agregarRestriccion(tipo : int):** agrega una nueva restricción indicando el tipo por medio de un número entero que se encuentra dentro del archivo sym.java. Valida que la restricción agregada en el momento indicado y donde permita que el problema logre culminar su solución de manera estable.

Ejecuta el método **agregarRestricción** del solucionador.

- **SimplexControlador:**
  - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con solucionar un problema de programación lineal, es una clase concreta de la clase AbstractControlador.
  - **UML:**



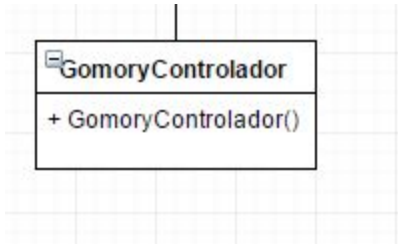
  - **Atributos:** no posee atributos únicamente los heredados de su superClase.
  - **Constructores:**
    - **SimplexControlador():** inicializa el parser y su solucionador como una instancia de SolucinadorSimplex.
  - **Métodos:** No sobrescribe ningún método de su superClase, ni implementa alguno nuevo.
- **MatrizControlador:**
  - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con reconocer una matriz numérica y realizar operaciones pivote en ella, es una clase concreta de la clase AbstractControlador..
  - **UML:**



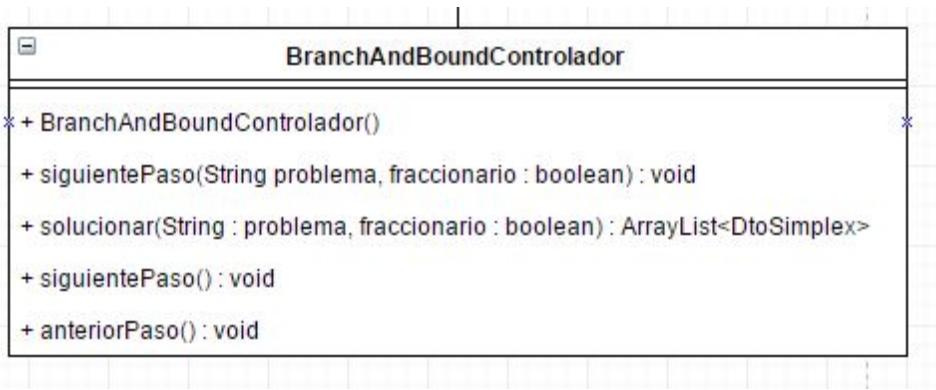
  - **Atributos:** no posee atributos únicamente los heredados de su superClase.
  - **Constructores:**
    - **SimplexControlador():** inicializa el parser y su solucionador como una instancia de SolucinadorSimplex.
  - Métodos: se describe los métodos que se sobreescriben.
    - **solucionar(String : problema, fraccionario : boolean):** inicializa el problema obtenido tras ingresar la matriz, no realiza la solución directa puesto no existe este método con este requerimiento.

- **siguientePaso(String problema, fraccionario : boolean):** inicializa la ejecución de la solución de una matriz por medio iteraciones intermedias. Ejecuta la validación de la matriz ingresada por medio del parser. Agrega el problema ingresado a la lista de pasos. Actualiza la vista con los datos creados.
- **siguientePaso():** itera sobre el elemento de la lista de pasos donde se encuentra el paso actual. Utiliza el método **siguientePaso** del solucionador. No se valida sobre los resultados de la iteración puesto que la matriz puede continuar operando sin importar las características que posee.Actualiza la vista con los datos de la siguiente operación.
- **anteriorPaso():** decrementa el contador de pasoActual. Actualiza la vista con los datos del paso anterior.
- **siguientesOperaciones(coordenada : Point) :** obtiene un String que representan a las siguientes operaciones sobre el elemento de la listaPasos donde encuentre el contador pasoActual. Dicho cadena de texto se obtiene por medio del método **siguientesOperaciones** del solucionador.
- **generarRadios(int Columna):** genera un arreglo de String con el valor representativo de los radios generados en la columna indicada, cada valor es correspondientemente por cada fila. Si se encuentre el infinito agrega un String representativo ya sea “oo” o “-oo”.
- **generarRadios() :** genera un arreglo de String con el valor representativo de los radios generados en la columna indicada dentro del punto que se encuentra en el paso actual de las iteraciones, cada valor es correspondientemente por cada fila. Si se encuentre el infinito agrega un String representativo ya sea “oo” o “-oo”.
- **modificarEntradaMatriz(fila : int, columna : int, valor : String) :** Cambia el valor de una entrada en la matriz de coeficientes actual, por un nuevo valor en la posición fila y columna indicado dentro de la iteración del paso actual.

        - **generarResumen():** recorre el arreglo de las iteraciones (**listaPasos**), ejecutando el método toString de cada DtoSimplex.
- **GomoryControlador:**
    - **Descripción:** Controlador concreto para los casos de uso que tienen que ver con solucionar un problema de programación lineal por medio del algoritmo de cortes de Gomory, es una clase concreta de la clase AbstractControlador.
    - **UML:**



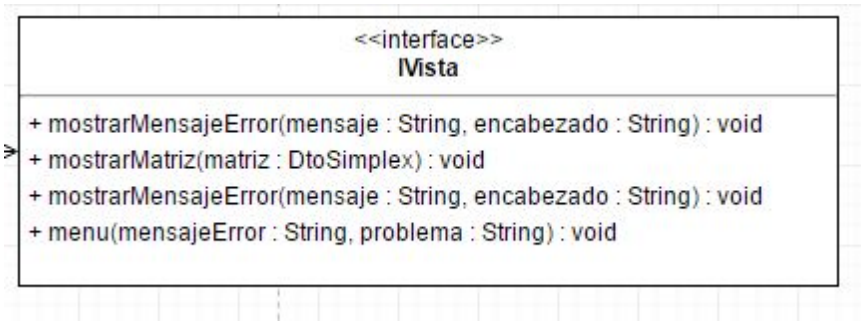
    - **Atributos:** no posee atributos únicamente los heredados de su superClase.
    - **Constructores:**
        - **GomoryControlador():** inicializa el parser y su solucionador como una instancia de SolucinadorGomory.
    - **Métodos:** No sobrescribe ningún método de su superClase, ni implementa alguno nuevo.
- **BranchAndBoundControlador:**
    - **Descripción**: Controlador concreto para los casos de uso que tienen que ver con solucionar un problema de programación lineal por medio del algoritmo de Branch and Bound, es una clase concreta de la clase AbstractControlador.
    - **UML:**



    - **Atributos:** no posee atributos únicamente los heredados de su superClase.

- **Constructores:**
  - **BranchAndBoundControlador():** inicializa el parser y su solucionador como una instancia de SolucionadorBranchAndBound.
- **Métodos:** se describen a continuación los metodos sobreescritos.
  - **solucionar(String : problema, fraccionario : boolean) :** Soluciona y obtiene inmediatamente todas las iteraciones del árbol del algoritmo de Branch and Bound. Indica el árbol generado en la solución y la información en el mensaje, cada uno dentro del DtoSimplex de la lista. Llena toda la lista de pasos al ejecutar. Actualiza la vista con la solución del problema.
  - **siguientePaso(String problema, fraccionario : boolean) :** este método soluciona y obtiene todas las iteraciones del árbol. Luego de ello apunta al contador pasoActual a la primer iteración para comenzar a navegar entre todos los pasos creados. Actualiza la vista.
  - **siguientePaso() :** aumenta el contador del paso actual y verifica que la siguiente iteración de la lista, tenga la solución del problema o el problema no sea factible. Actualiza la vista con los datos de la siguiente iteración.
  - **anteriorPaso():** decrementa el contador de pasoActual. Actualiza la vista con los datos del paso anterior.

- **IVista:**
  - **Descripción:** Interfaz que define los métodos que necesita una clase de tipo vista para desplegar la información solicitada por el cliente. .
  - **UML:**



- **Atributos:** esta interfaz no posee atributos.
- **Métodos**:
  - **mostrarMensajeError(mensaje : String, encabezado : String) :** muestra un mensaje de error en la vista que lo implemente. El mensaje y el encabezado deben ser indicados.
  - **mostrarMatriz(matriz : DtoSimplex) :** toma la matriz de AbstractFraccion dentro del DtoSimplex, y utilizando el toString con un formato indicado lo muestra en la vista que implemente este método.
  - **mostrarMensajeError(mensaje : String, encabezado : String):** muestra un mensaje de información en la vista que lo implemente. El mensaje y el encabezado deben ser indicados.
  - **menu(mensajeError : String, problema : String) :** método usado para cuando sucede un error de sintaxis en la cadena de texto ingresado. Lanza un mensaje de error y muestra el problema ingresado en la pantalla inicial.

5.2.3. Vista

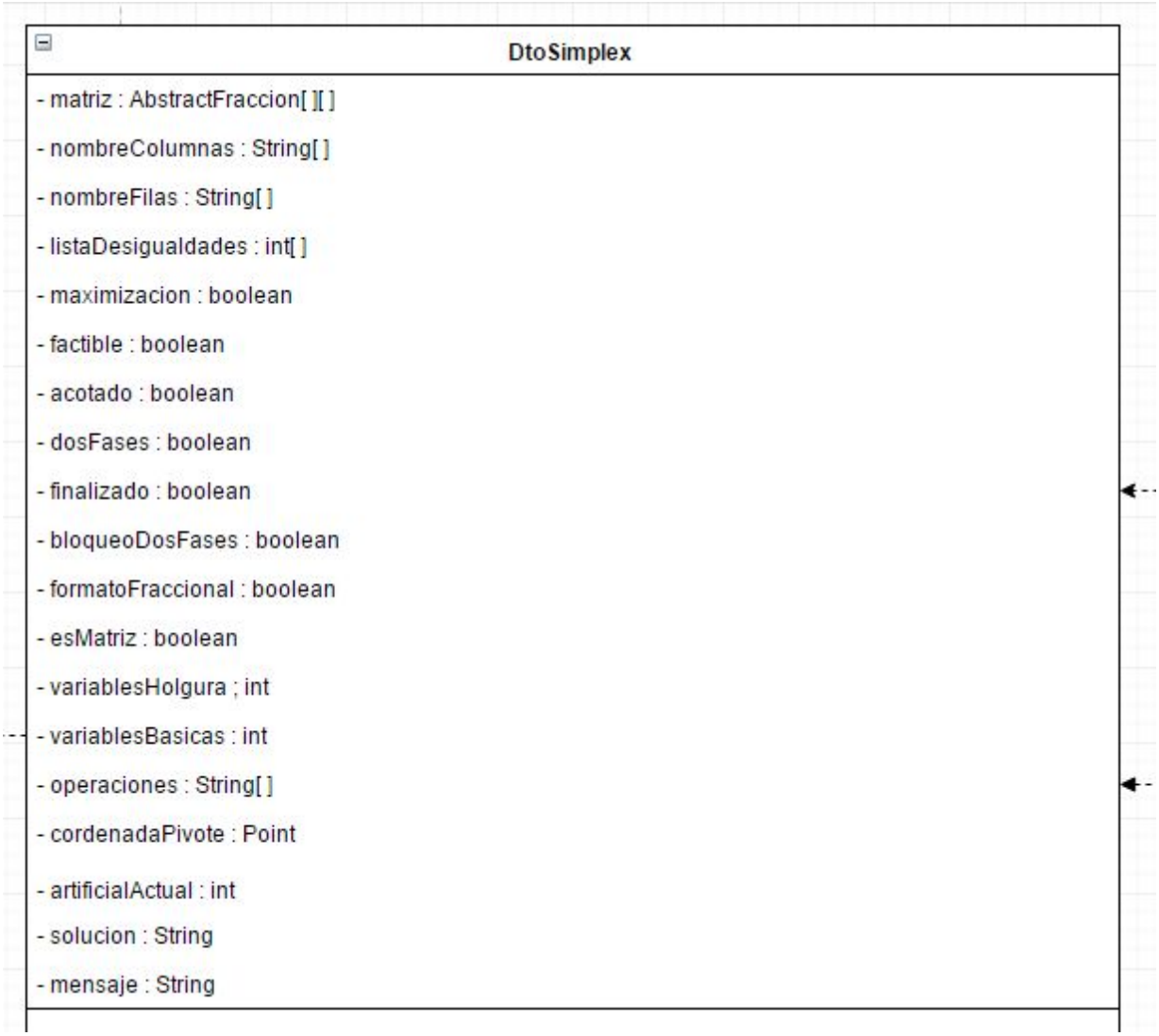
Contiene las dos pantallas de interfaces gráficas de la solución: PantallaPrincipal y PantallaPasoIntermedio.

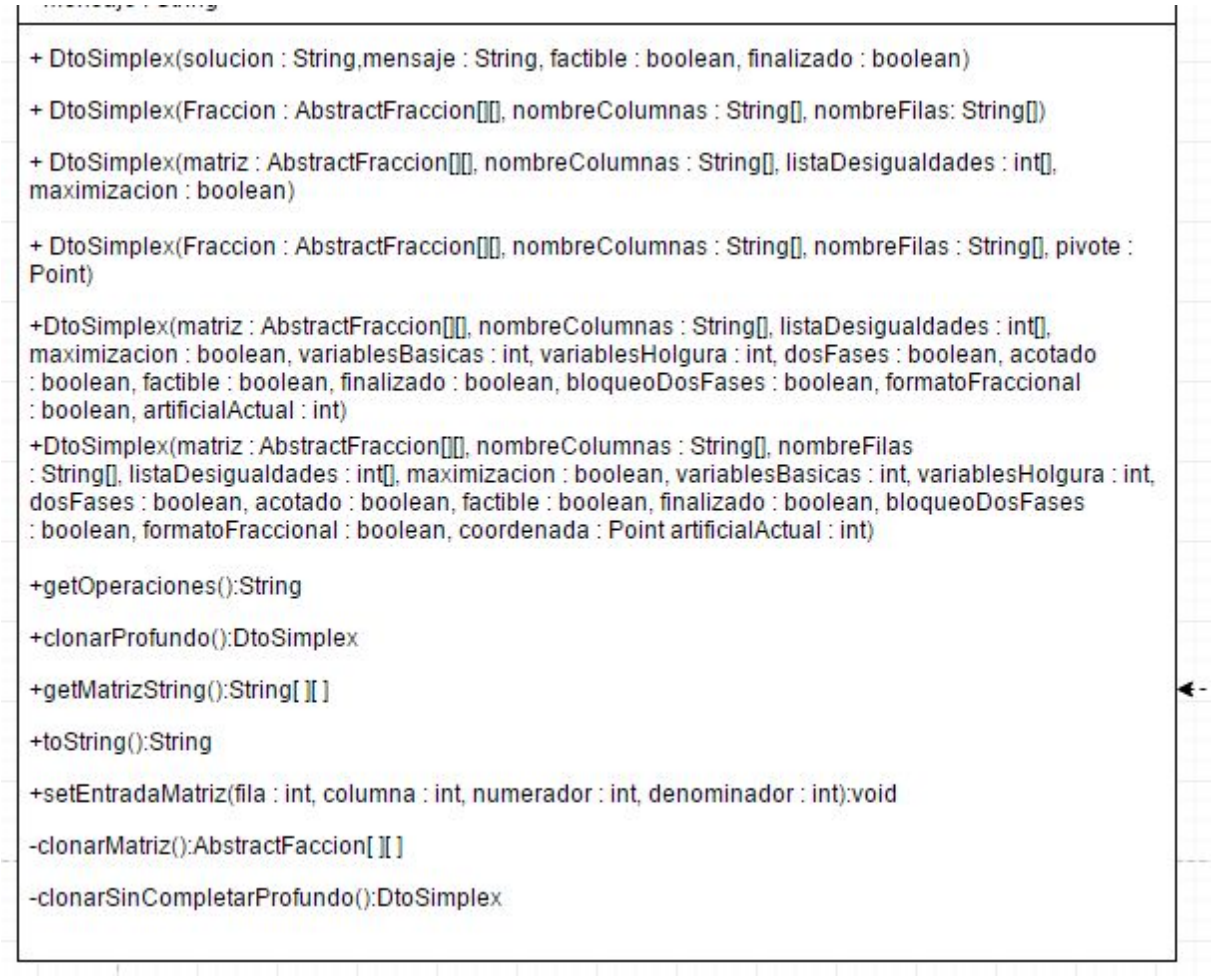
- **PantallaPrincipal:**
  - **Descripción:** Pantalla principal y punto de entrada del programa. Ella es la encargada de recibir el problema de programación lineal por parte del usuario e instanciar el controlador respectivo.
- **PantallaPasoIntermedio:**
  - **Descripción:** Pantalla encargada de mostrar 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 actualiza la vista de manera respectiva.
- **PantallaPasoIntermedioBranchAndBound:**
  - **Descripción:** Pantalla encargada de mostrar al usuario el estado actual de la solución del problema con el algoritmo de BranchAndBound .Ella tiene lógica para enviarle al controlador las acciones que realiza el usuario, y que de esta manera el controlador actualiza la vista de manera respectiva.

5.2.4. Dto

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

- **UML:**


DtoSimplex
- matriz : AbstractFraccion[ ][ ]
- nombreColumnas : String[ ]
- nombreFilas : String[ ]
- listaDesigualdades : int[ ]
- maximizacion : boolean
- factible : boolean
- acotado : boolean
- dosFases : boolean
- finalizado : boolean
- bloqueoDosFases : boolean
- formatoFraccional : boolean
- esMatriz : boolean
- variablesHolgura ; int
- variablesBasicas : int
- operaciones : String[ ]
- cordenadaPivote : Point
- artificialActual : int
- solucion : String
- mensaje : String



- **Atributos:**
  - **matriz : AbstractFraccion[ ][ ]:** matriz de elementos tipo AbstractFraccion cual representa al problema de programación mientras se encuentra en solución.
  - **nombreColumnas : String[]:** nombre de las variables a la cual se le asigna un columna de la **matriz.**
  - **nombreFilas : String[ ]**: nombre de las variables a la cual se le asigna un fila de la **matriz**.
  - **listaDesigualdades : int[]:** lista de desigualdades representadas por un número entero que contiene cada restricción del problema de programación lineal.
  - **maximizacion : boolean:** Valor binario que indica si el problema se encuentra solucionando un problema de maximización (**true**) o de minimización (**false**).

- **factible : boolean:** indica si el problema de programación lineal contenido dentro, es factible (**true**) o no (**false**).
- **acotado : boolean:** indica si el problema de programación contenido dentro, esta acotado (**true**) o no (**false**).
- **dosFases : boolean:** indica si el problema de programación contenido dentro, se tuvo que solucionar por el método de dos fases y se encuenta en la primera etapa (**true**) o no (**false**).
- **finalizado : boolean:** muestra si el problema esta finalizado (**true**) o no(**false**).
- **bloqueoDosFases : boolean:** indica al inicio del método Simplex si el problema se encuentra eliminando las variables artificiales cuando se soluciona por medio de dos fases. Si es **true** se encuentra eliminando los unos de la función w y de lo contrario **false**.
- **formatoFraccional : boolean:** brinda el formato de salida de las fracciones. Si es **true** usa un formato fraccional y **false** usa decimal.
- **esMatriz : boolean:** si la matriz de fracciones fue ingresada por tipo matriz entonces tendrá un valor de true y false si fue ingresada por medio del formato de programación lineal.
- **variablesHolgura : int:** cantidad de variables de holgura del problema de programación lineal.
- **variablesBasicas : int:** cantidad de variables básicas contenida por el problema original.
- **operaciones : String[ ]:** arreglo de elementos tipo String, donde se contiene las siguientes operaciones filas que se realizarán según la coordenada del objeto.
- **cordenadaPivote : Point:** punto con un valor **x** (columna) y **y** (fila), que significa donde se realizará la operación fila.
- **artificialActual : int:** índice de la columna donde se encuentra una variable artificial por eliminar cuando existe el bloque de dos fases.
- **solucion : String:** String con la solución del problema, contiene el valor de cada variable.

    - **mensaje : String :** String con información de lo sucedido en el algoritmo.
- **Constructores:**
    - **DtoSimplex(solucion : String,mensaje : String, factible : boolean, finalizado : boolean)**
    - **DtoSimplex(Fraccion : AbstractFraccion[][], nombreColumnas : String[], nombreFilas: String[]):** inicializa una instancia un objeto de transferencia de datos con la atributos solución, mensaje, factible y finalizado con un valor indicado.
    - **DtoSimplex(Fraccion : AbstractFraccion[][], nombreColumnas : String[], nombreFilas: String[]):** inicializa una instancia un objeto de transferencia de datos con atributos matriz, nombreColumna y nombreFila indicados.
    - **+ DtoSimplex(Fraccion : AbstractFraccion[][], nombreColumnas : String[], nombreFilas : String[], pivote : Point):** inicializa una instancia un objeto de transferencia de datos conlos atributos matriz, nombreColumna, nombreFila y coordenada indicados.
    - **+DtoSimplex(matriz : AbstractFraccion[][], nombreColumnas : String[], listaDesigualdades : int[], maximizacion : boolean, variablesBasicas : int, variablesHolgura : int, dosFases : boolean, acotado : boolean, factible : boolean, finalizado : boolean, bloqueoDosFases : boolean, formatoFraccional : boolean, artificialActual : int):** inicializa una instancia un objeto de transferencia de datos con los atributos matriz, nombreColumnas, listaDesigualdades, maximizacion, variablesBasicas, variablesHolgura, dosFases, acotado, factible, finalizado, bloqueoDosfases, formatoFraccional y artificialActual indicados.
    - **+DtoSimplex(matriz : AbstractFraccion[][], nombreColumnas : String[], nombreFilas : String[], listaDesigualdades : int[], maximizacion : boolean, variablesBasicas : int, variablesHolgura : int, dosFases : boolean, acotado : boolean, factible : boolean, finalizado : boolean, bloqueoDosFases : boolean,**

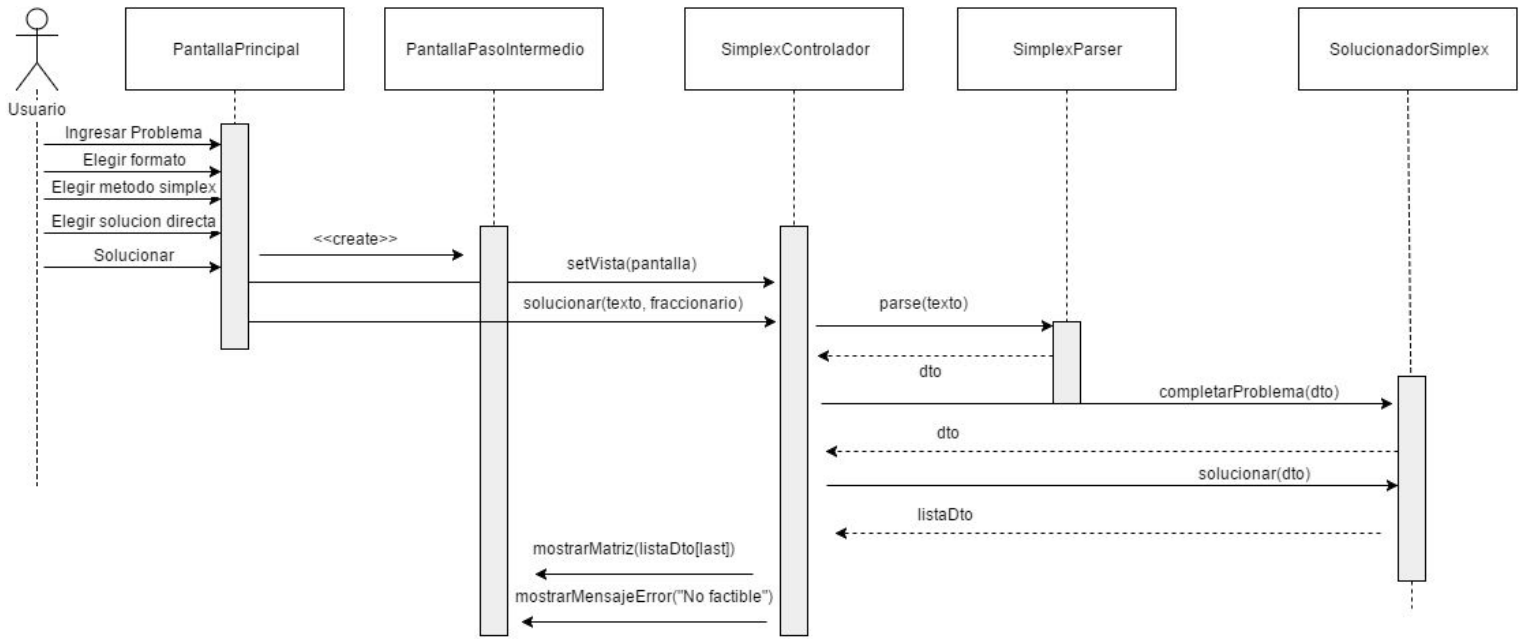
formatoFraccional : boolean, coordenada : Point artificialActual : int): inicializa una instancia un objeto de transferencia de datos con los atributos matriz, nombreColumnas, nombreFilas, listaDesigualdades, maximizacion, variablesBasicas, variablesHolgura, dosFases, acotado, factible, finalizado, bloqueoDosfases, formatoFraccional, coordenada y artificialActual indicados.

- **Métodos:**
  - **getOperaciones():** retorna los elementos del arreglo de String operaciones, agregados todos en una sola cadena separados por saltos de líneas.
  - **clonarProfundo():** copia todos los atributos del DtoSimplex y retorna una nueva referencia para el objeto y sus atributos.
  - **getMatrizString():** retorna una matriz de elementos tipo String, donde se agregan los elementos del atributo **matriz** ejecutando el método toString de cada Fraccion con el formato indicado en el tributo **formatoFraccional**.
  - **toString():** retorna un String con el valor de cada Fraccion del atributo matriz dentro de un marco con caracteres '-' conjunto al nombre de las filas y columnas. Ambos atributos de la clase.
  - **setEntradaMatriz(fila : int, columna : int, numerador : int, denominador : int):** actualiza el valor de la **matriz** en la fila y columna indicada. Dicho valor que se agrega es tomado de los parámetros numerador y denominador encapsulados en una instancia tipo AbstractFraccion.
  - **clonarMatriz():** Crea una nueva matriz de elementos tipo abstractFraccion donde cada dato posee una diferente referencia a su origen. El origen de esta nueva **matriz** es el atributo matriz.
  - **clonarSinCompletarProfundo():** copia todos los atributos del DtoSimplex, excepto coordenada y NombresFilas, y retorna una nueva referencia para el objeto y sus atributos.

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. Asimismo se describirá cada diagrama, explicando la manera en que la funcionalidad fue implementada.

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

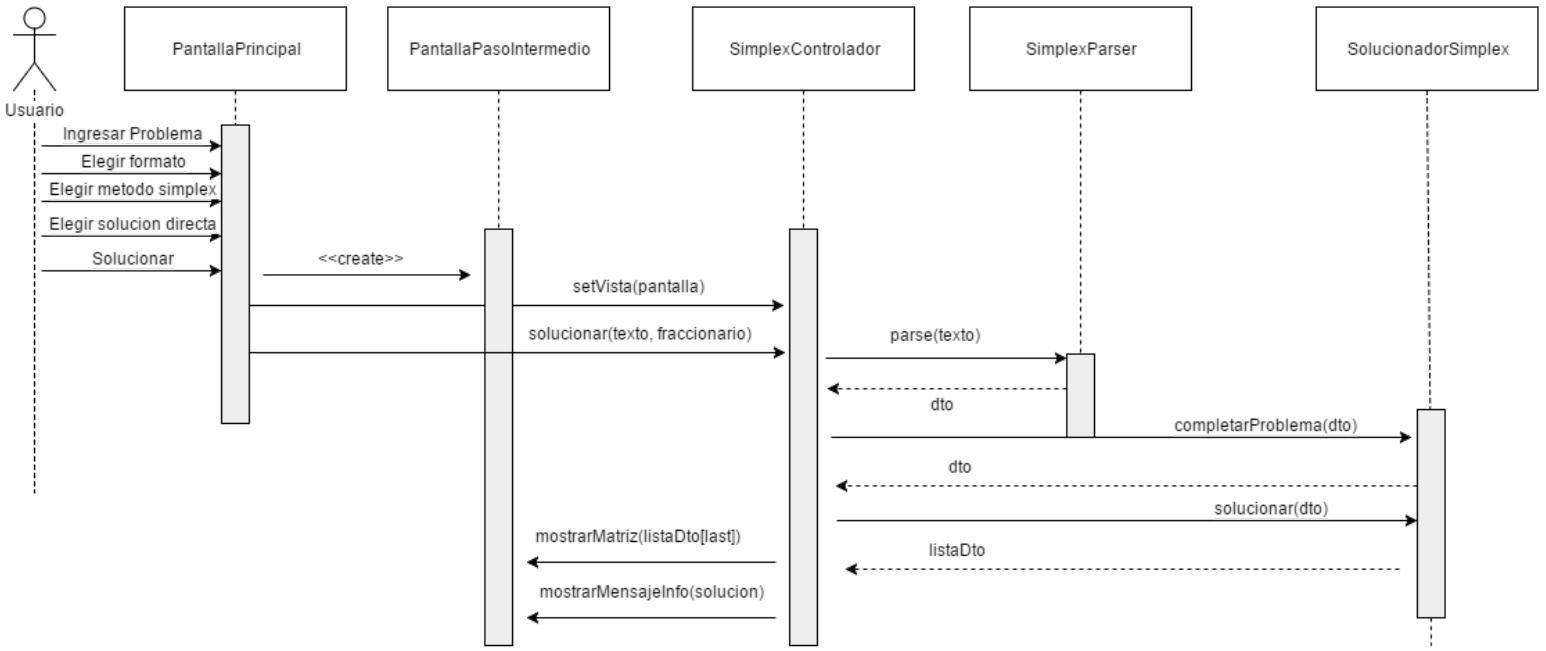


6.1.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.

7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
8. PantallaPrincipal realiza inyección de dependencia a la clase SimplexControlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que SimplexControlador pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a SimplexControlador que el usuario ha solicitado solucionar un problema de manera directa, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.
10. El hilo de ejecución es pasado a SimplexControlador. Esta clase es el centro de control de la lógica.
11. SimplexControlador toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
12. SimplexParser retorna el DTO a SimplexControlador.
13. Una vez con el problema textual representado por medio de un DTO, SimplexControlador envía esta información a SolucionadorSimplex para encontrar la respuesta del problema.
14. SolucionadorSimplex completa el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a SimplexControlador.
15. Una vez que el DTO se encuentra completo, SimplexControlador le indica a SolucionadorSimplex que solucione el problema de manera directa, retornando la solución en forma de DTO a SimplexControlador.
16. SimplexControlador actualiza la vista por medio del método mostrarMatriz y muestra el resultado obtenido en PantallaPasoIntermedio.
17. Si el problema no fuera factible, SimplexControlador mostrará un mensaje de error mediante el metodo mostrarMensajeError.

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



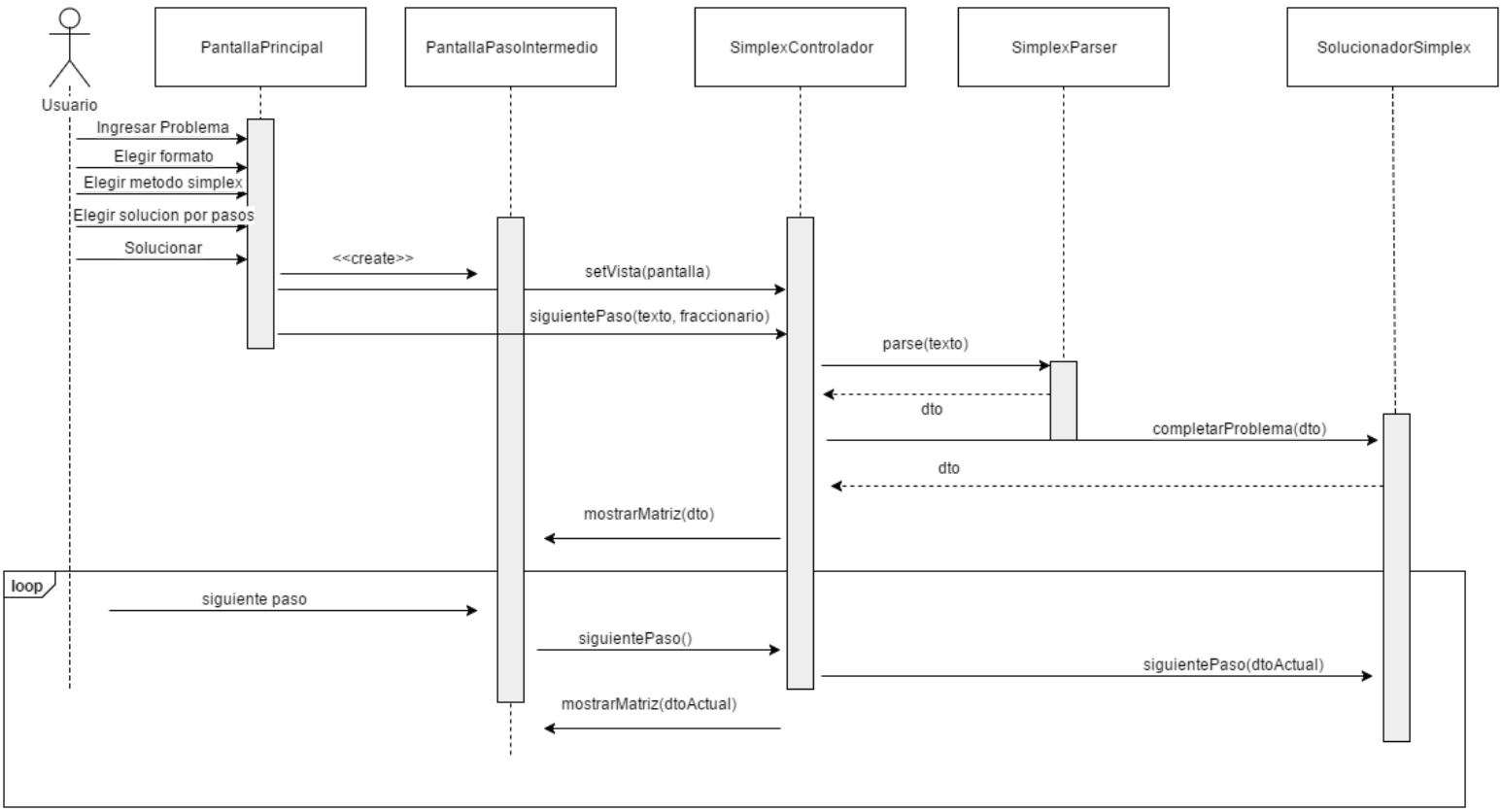
6.2.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.
7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
8. PantallaPrincipal realiza inyección de dependencia a la clase SimplexControlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que SimplexControlador pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a SimplexControlador que el usuario ha solicitado solucionar un problema de manera directa, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.
10. El hilo de ejecución es pasado a SimplexControlador. Esta clase es el centro de control de la lógica.
11. SimplexControlador toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO)

que será utilizado para pasar información durante el resto de la ejecución.

12. SimplexParser retorna el DTO a SimplexControlador.
13. Una vez con el problema textual representado por medio de un DTO, SimplexControlador envía esta información a SolucionadorSimplex para encontrar la respuesta del problema.
14. SolucionadorSimplex completa el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a SimplexControlador.
15. Una vez que el DTO se encuentra completo, SimplexControlador le indica a SolucionadorSimplex que solucione el problema de manera directa, retornando la solución en forma de DTO a SimplexControlador.
16. SimplexControlador actualiza la vista por medio del método mostrarMatriz y muestra el resultado obtenido en PantallaPasoIntermedio.

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

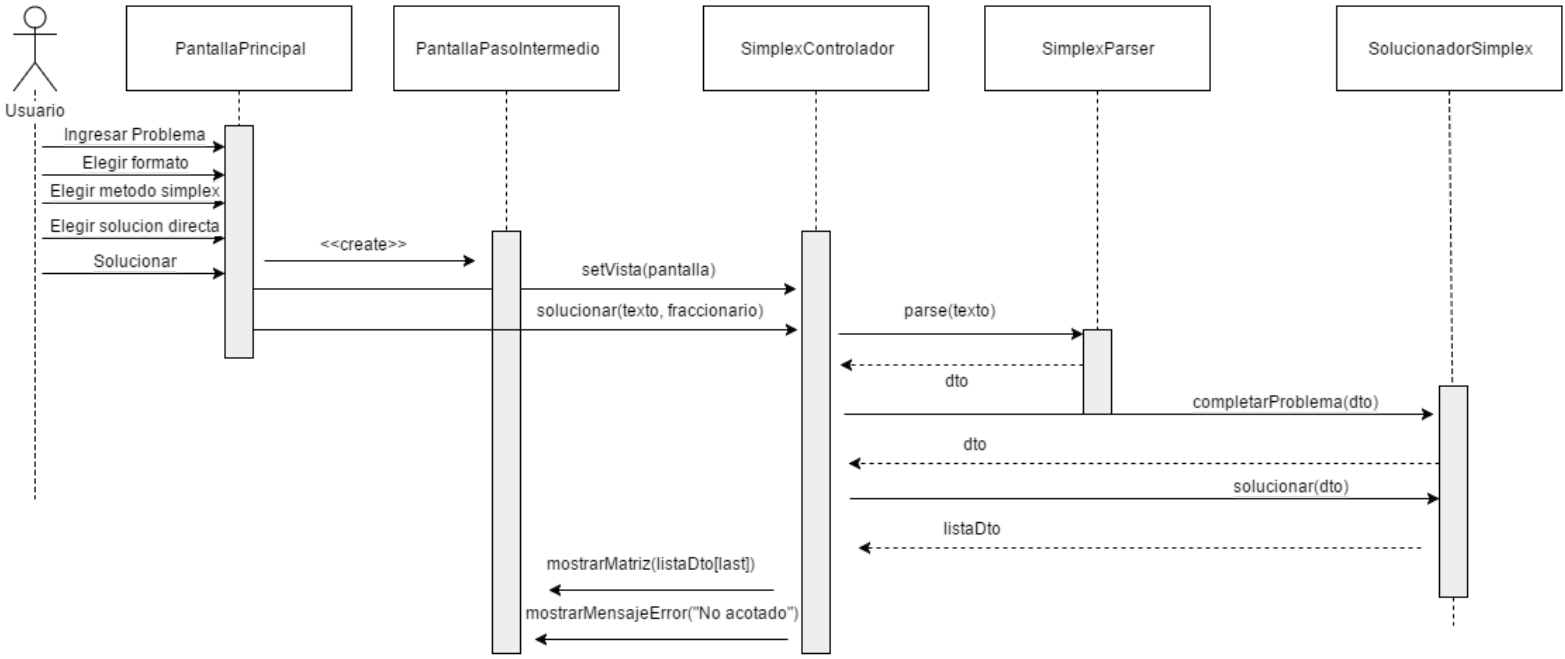


6.3.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.
7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
8. PantallaPrincipal realiza inyección de dependencia a la clase SimplexControlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que SimplexControlador pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a SimplexControlador que el usuario ha solicitado solucionar un problema paso por paso, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.
10. El hilo de ejecución es pasado a SimplexControlador. Esta clase es el centro de control de la lógica.
11. SimplexControlador toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
12. SimplexParser retorna el DTO a SimplexControlador.
13. Una vez con el problema textual representado por medio de un DTO, SimplexControlador envía esta información a SolucionadorSimplex para encontrar la respuesta del problema.
14. SolucionadorSimplex completa el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a SimplexControlador.
15. Una vez que el DTO se encuentra completo, SimplexControlador le indica a SolucionadorSimplex que muestre el siguiente paso del algoritmo, es decir, la siguiente iteración.
16. SimplexControlador actualiza la vista por medio del método mostrarMatriz y muestra el siguiente paso en PantallaPasoIntermedio.

17. PantallaPasoIntermedio muestra la matriz que representa el problema al usuario.
18. El usuario indica a PantallaPasointermedio que desea conocer el siguiente paso.
    a. Mientras no se haya llegado a un óptimo y el problema sea factible y acotado, volver al punto 15.
19. Cuando se ha llegado a un óptimo, PantallaPasoIntermedio lo indica por medio de un mensaje y termina la ejecución.

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



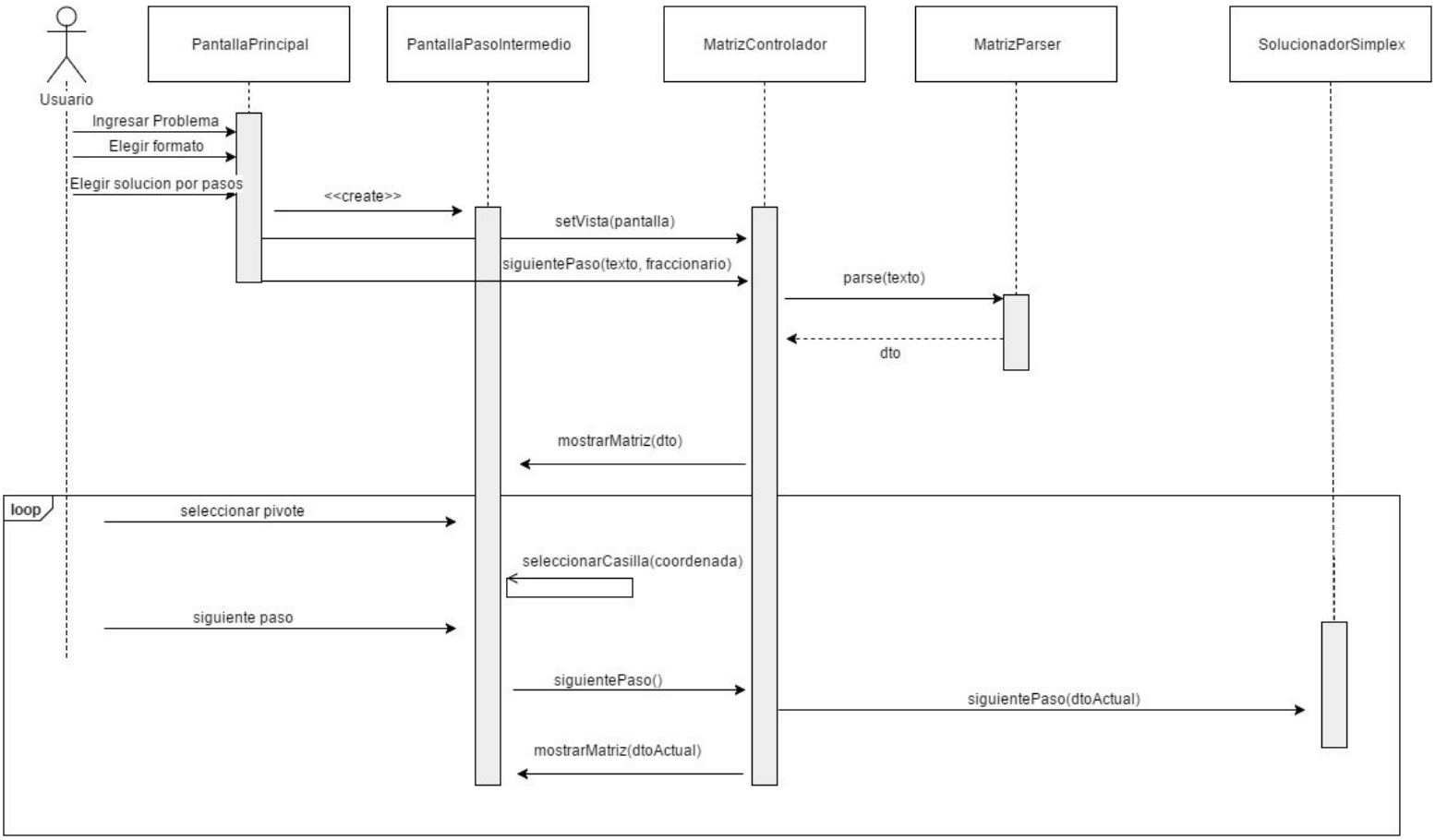
6.4.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.
7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
8. PantallaPrincipal realiza inyección de dependencia a la clase SimplexControlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que SimplexControlador pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a SimplexControlador que el usuario ha solicitado solucionar un problema de manera directa, y envía el

texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.

10. El hilo de ejecución es pasado a SimplexControlador. Esta clase es el centro de control de la lógica.
11. SimplexControlador toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
12. SimplexParser retorna el DTO a SimplexControlador.
13. Una vez con el problema textual representado por medio de un DTO, SimplexControlador envía esta información a SolucionadorSimplex para encontrar la respuesta del problema.
14. SolucionadorSimplex completa el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a SimplexControlador.
15. Una vez que el DTO se encuentra completo, SimplexControlador le indica a SolucionadorSimplex que solucione el problema de manera directa, retornando la solución en forma de DTO a SimplexControlador.
16. SimplexControlador actualiza la vista por medio del método mostrarMatriz y muestra el resultado obtenido en PantallaPasoIntermedio.
17. Si el problema no estuviera acotado, SimplexControlador mostrará un mensaje de error mediante el metodo mostrarMensajeError.

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

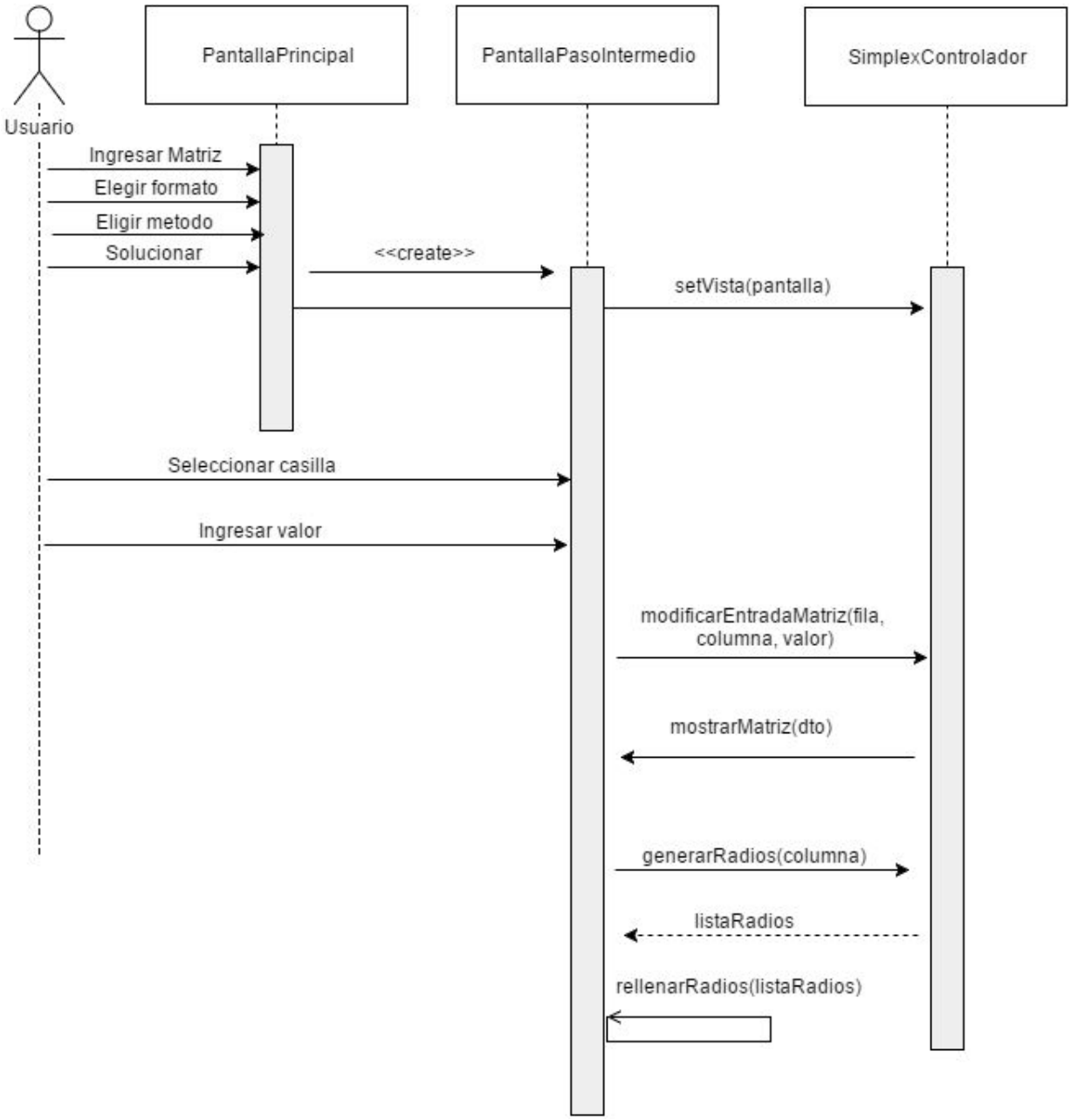


6.5.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa la representación matricial que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Matriz Numérica para solucionar el problema en PantallaPrincipal.
5. El usuario da clic en el botón solucionar.
6. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
7. PantallaPrincipal realiza inyección de dependencia a la clase MatrizControlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que MatrizControlador pueda actualizar la vista de manera acorde.
8. PantallaPrincipal indica a MatrizControlador que el usuario ha solicitado solucionar una matriz, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.
9. El hilo de ejecución es pasado a MatrizControlador. Esta clase es el centro de control de la lógica.
10. MatrizControlador toma el problema textual ingresado y mediante la clase MatrizParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
11. MatrizParser retorna el DTO a MatrizControlador.
12. MatrizControlador actualiza la vista por medio del método mostrarMatriz y muestra la matriz en PantallaPasoIntermedio.
13. El usuario indica mediante el uso de su mouse la casilla de la matriz donde desea realizar un pivoteo.
14. PantallaPasoIntermedio actualiza la casilla seleccionada en el DTO que representa la matriz actual.
15. El usuario hace clic en el botón “Siguiente paso” para pivotar.
16. PantallaPasoIntermedio indica a MatrizControlador que calcule el siguiente paso. Dentro del DTO se encuentra la matriz actual junto con la coordenada que será utilizada como pivote.
17. MatrizControlador utiliza SolucionadorSimplex para pivotar en la posición indicada. SolucionadorSimplex retorna el DTO con la nueva matriz.

18. MatrizControlador muestra la nueva matriz al usuario por medio de PantallaPasoIntermedio.
19. Si el usuario desea continuar pivotando, regresa al punto 13.

6.6. Modificar la entrada de una matriz

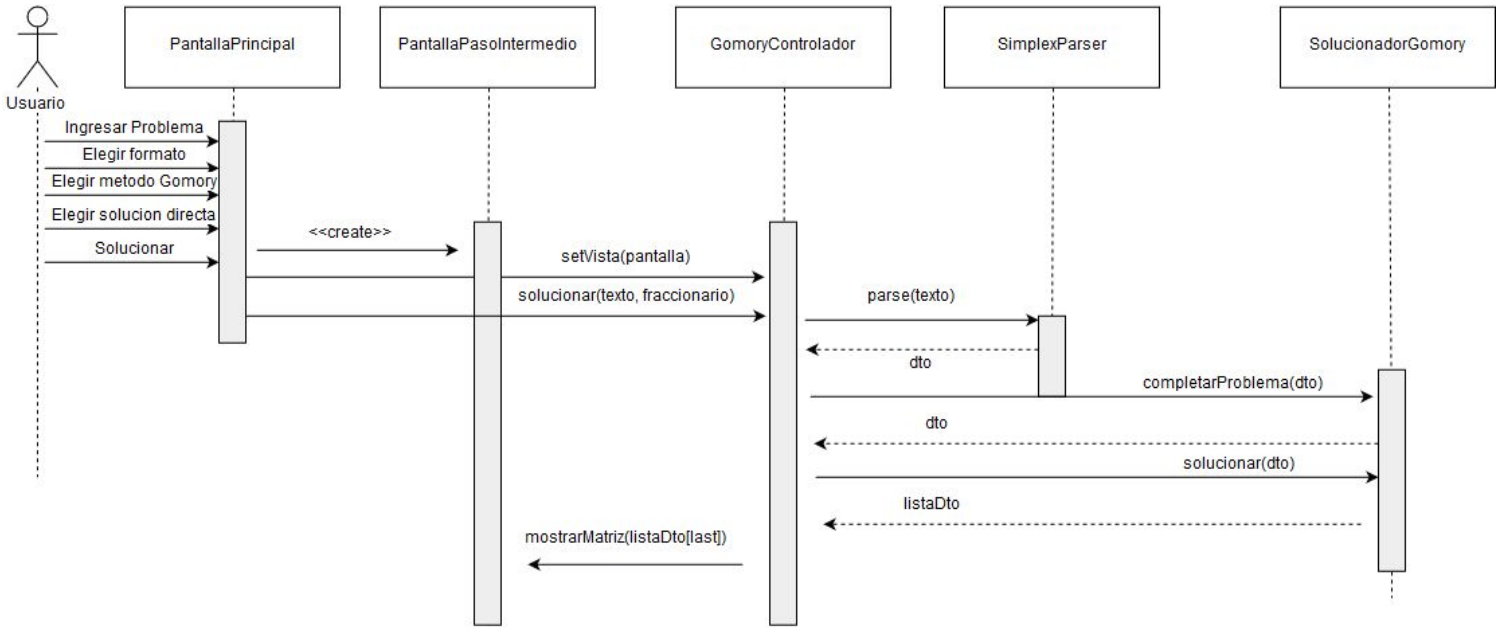


6.6.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.

2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa la representación matricial o el problema que desea resolver en PantallaPrincipal.
4. El usuario escoge el método que desee para solucionar el problema en PantallaPrincipal.
5. El usuario da clic en el botón solucionar.
6. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
7. PantallaPrincipal realiza inyección de dependencia al controlador respectivo, y pasa una referencia a la recién creada PantallaPasoIntermedio para que el controlador pueda actualizar la vista de manera acorde.
8. El controlador actualiza la vista con el la representación matricial respectiva.
9. El usuario por medio del mouse hace clic en la casilla que desea modificar.
10. El usuario utiliza el teclado numérico para actualizar el valor, y presiona la tecla Enter cuando ha terminado.
11. PantallaPasoIntermedio indica al controlador que el usuario ha modificado una matriz, e invoca al método modificarEntradaMatriz con los parámetros necesarios. El controlador muestra la nueva matriz en PantallaPasoIntermedio, seguidamente actualiza el valor de los radios mostrados en pantalla, rellenando los radios mostrados de manera acorde.

6.7. Obtener la solución entera óptima de un problema de programación lineal mediante el algoritmo de Cortes de Gomory de manera directa.

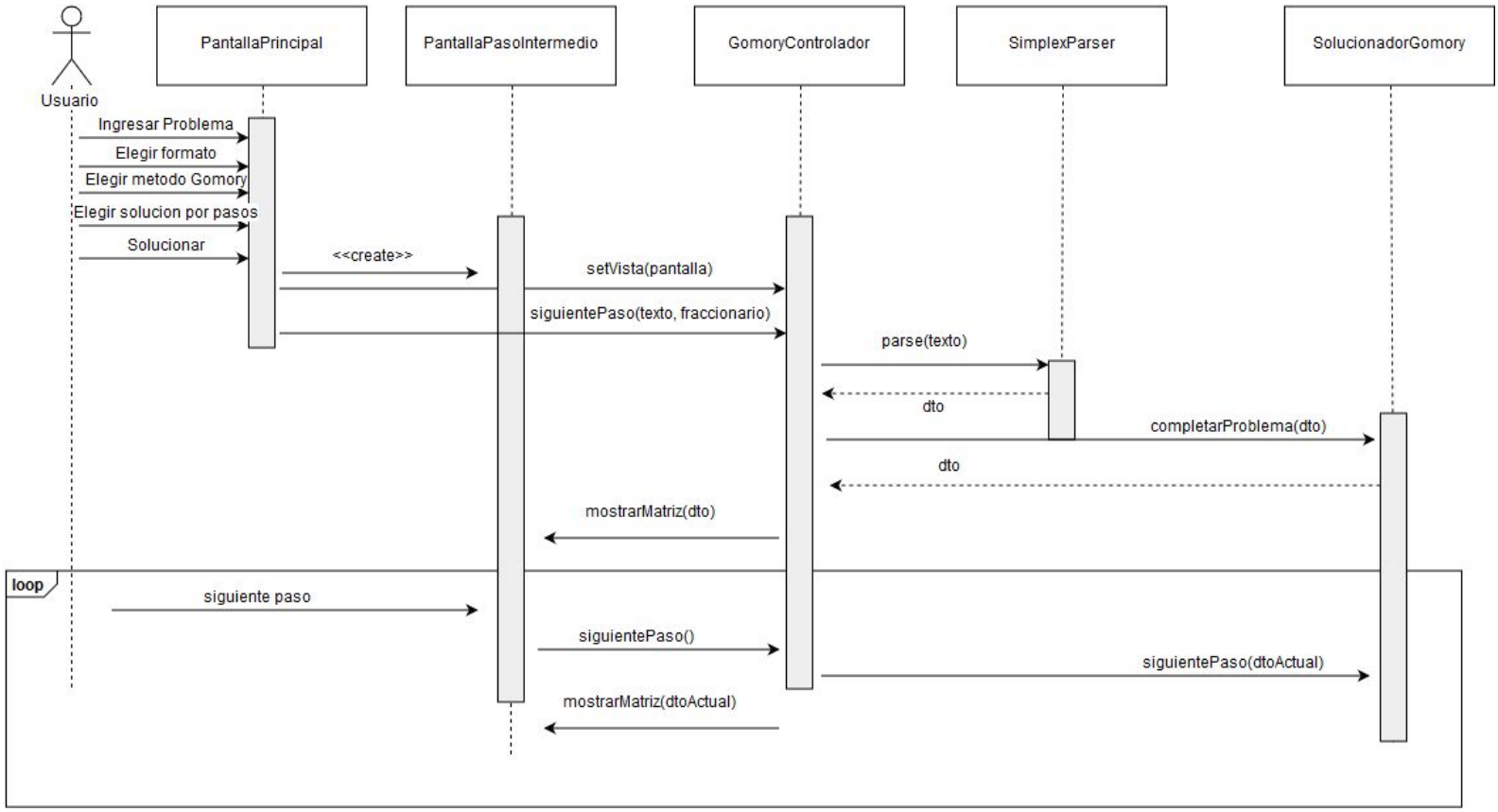


6.7.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.
7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
8. PantallaPrincipal realiza inyección de dependencia a la clase GomoryControlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que GomoryControlador pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a GomoryControlador que el usuario ha solicitado solucionar un problema entero de manera directa, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.

10. El hilo de ejecución es pasado a GomoryControlador. Esta clase es el centro de control de la lógica.
11. GomoryControlador toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
12. SimplexParser retorna el DTO a GomoryControlador.
13. Una vez con el problema textual representado por medio de un DTO, GomoryControlador envía esta información a SolucionadorGomory para encontrar la respuesta del problema.
14. SolucionadorGomory completa el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a GomoryControlador.
15. Una vez que el DTO se encuentra completo, GomoryControlador le indica a SolucionadorGomory que solucione el problema de manera directa, retornando el resultado en forma de DTO a GomoryControlador.
16. GomoryControlador actualiza la vista por medio del método mostrarMatriz y muestra el resultado obtenido en PantallaPasoIntermedio.

6.8. Obtener la solución entera óptima de un problema de programación lineal mediante el algoritmo de Cortes de Gomory mostrando los pasos intermedios.

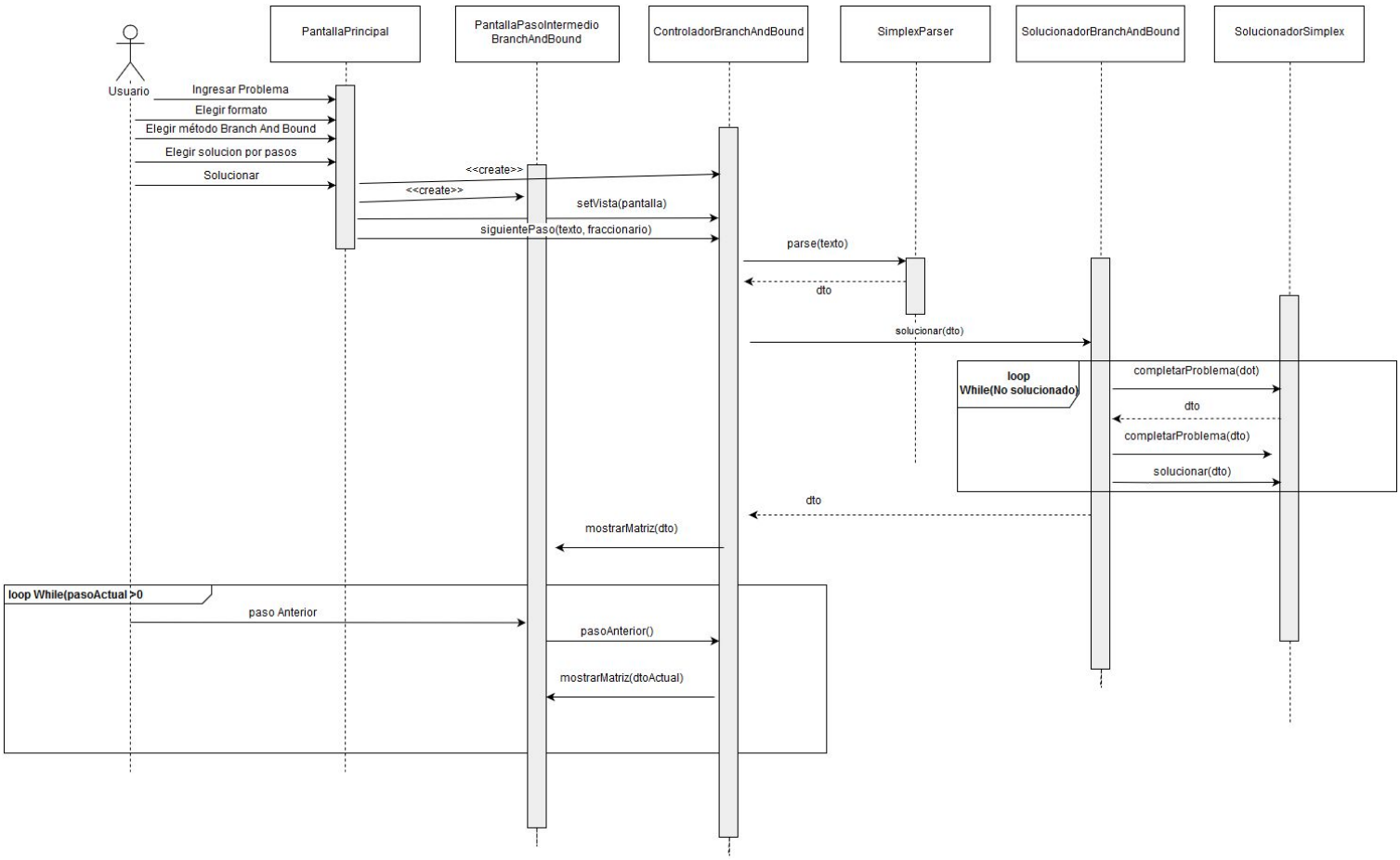


6.8.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.
7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.

8. PantallaPrincipal realiza inyección de dependencia a la clase GomoryControlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que GomoryControlador pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a GomoryControlador que el usuario ha solicitado solucionar un problema entero paso por paso, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.
10. El hilo de ejecución es pasado a GomoryControlador. Esta clase es el centro de control de la lógica.
11. GomoryControlador toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
12. SimplexParser retorna el DTO a GomoryControlador.
13. Una vez con el problema textual representado por medio de un DTO, GomoryControlador envía esta información a SolucionadorGomory para encontrar la respuesta del problema.
14. SolucionadorGomory completa el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a GomoryControlador.
15. Una vez que el DTO se encuentra completo, GomoryControlador le indica a SolucionadorGomory que muestre el siguiente paso del algoritmo, es decir, la siguiente iteración.
16. GomoryControlador actualiza la vista por medio del método mostrarMatriz y muestra el siguiente paso en PantallaPasoIntermedio.
17. PantallaPasoIntermedio muestra la matriz que representa el problema al usuario.
18. El usuario indica a PantallaPasointermedio que desea conocer el siguiente paso.
    a. Mientras no se haya llegado a un óptimo y el problema sea factible y acotado, volver al punto 15.
19. Cuando se ha llegado a un óptimo, PantallaPasoIntermedio lo indica por medio de un mensaje y termina la ejecución.

6.9. Obtener la solución entera óptima de un problema de programación lineal mediante el algoritmo de Branch and Bound de manera directa.

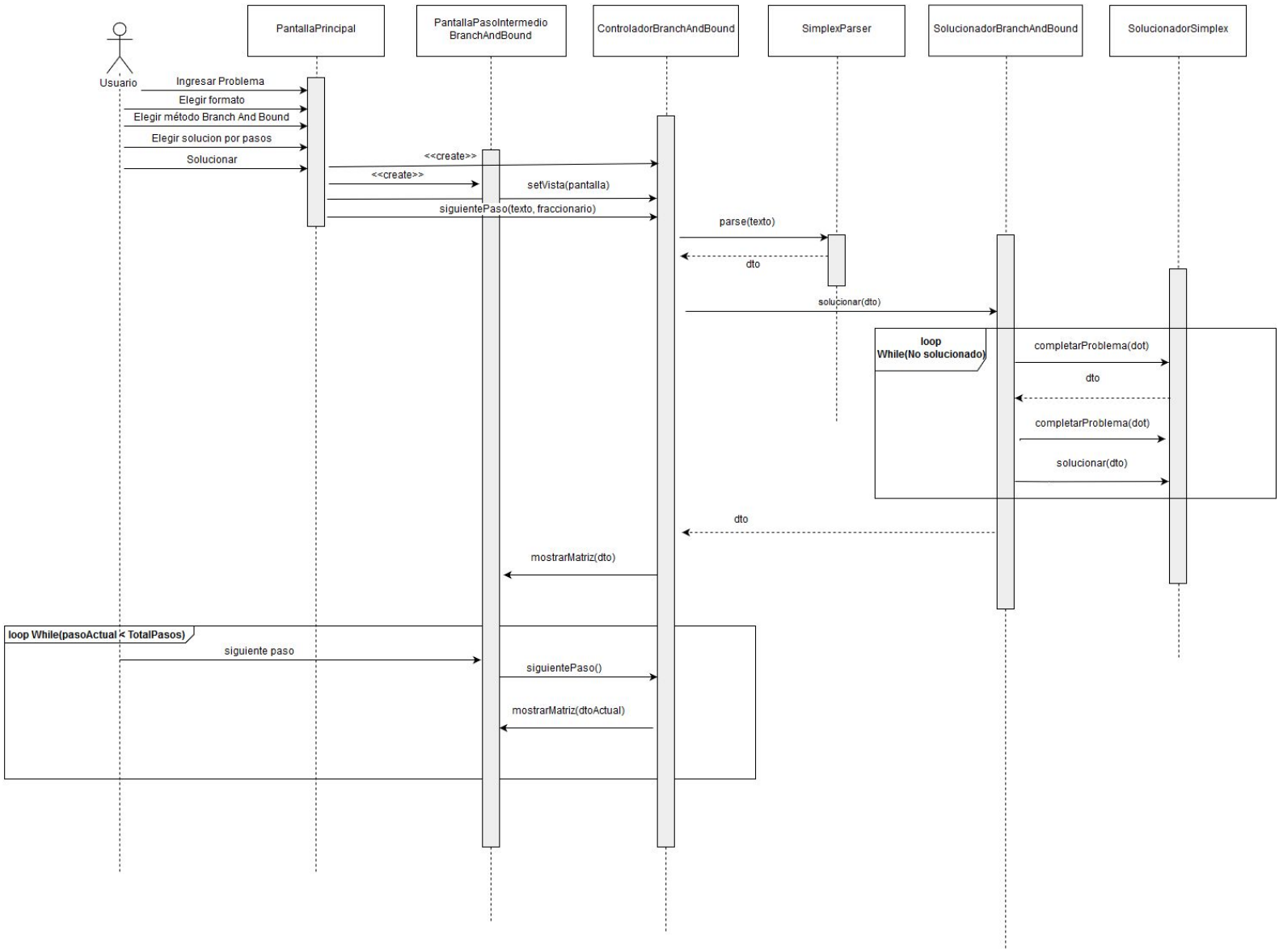


6.9.1. Flujo de Ejecución

1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.
4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.
7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.

8. PantallaPrincipal realiza inyección de dependencia a la clase Controlador, y pasa una referencia a la recién creada PantallaPasoIntermedio para que ControladorBranchAndBound pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a ControladorBranchAndBound que el usuario ha solicitado solucionar un problema entero de manera directa, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.
10. El hilo de ejecución es pasado a ControladorBranchAndBound. Esta clase es el centro de control de la lógica.
11. ControladorBranchAndBound toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
12. SimplexParser retorna el DTO a ControladorBranchAndBound.
13. Una vez con el problema textual representado por medio de un DTO, ControladorBranchAndBound envía esta información a SolucionadorBranchAndBound para encontrar la respuesta del problema.
14. SolucionadorBranchAndBound completa el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a ControladorBranchAndBound.
15. Una vez que el DTO se encuentra completo, ControladorBranchAndBound le indica a SolucionadorBranchAndBound que solucione el problema de manera directa, retornando el resultado en forma de DTO a ControladorBranchAndBound.
16. ControladorBranchAndBound actualiza la vista por medio del método mostrarMatriz y muestra el resultado obtenido en PantallaPasoIntermedio.

6.10. Obtener la solución entera óptima de un problema de programación lineal mediante el algoritmo de Branch and Bound mostrando los pasos intermedios.



6.10.1. Flujo de Ejecución

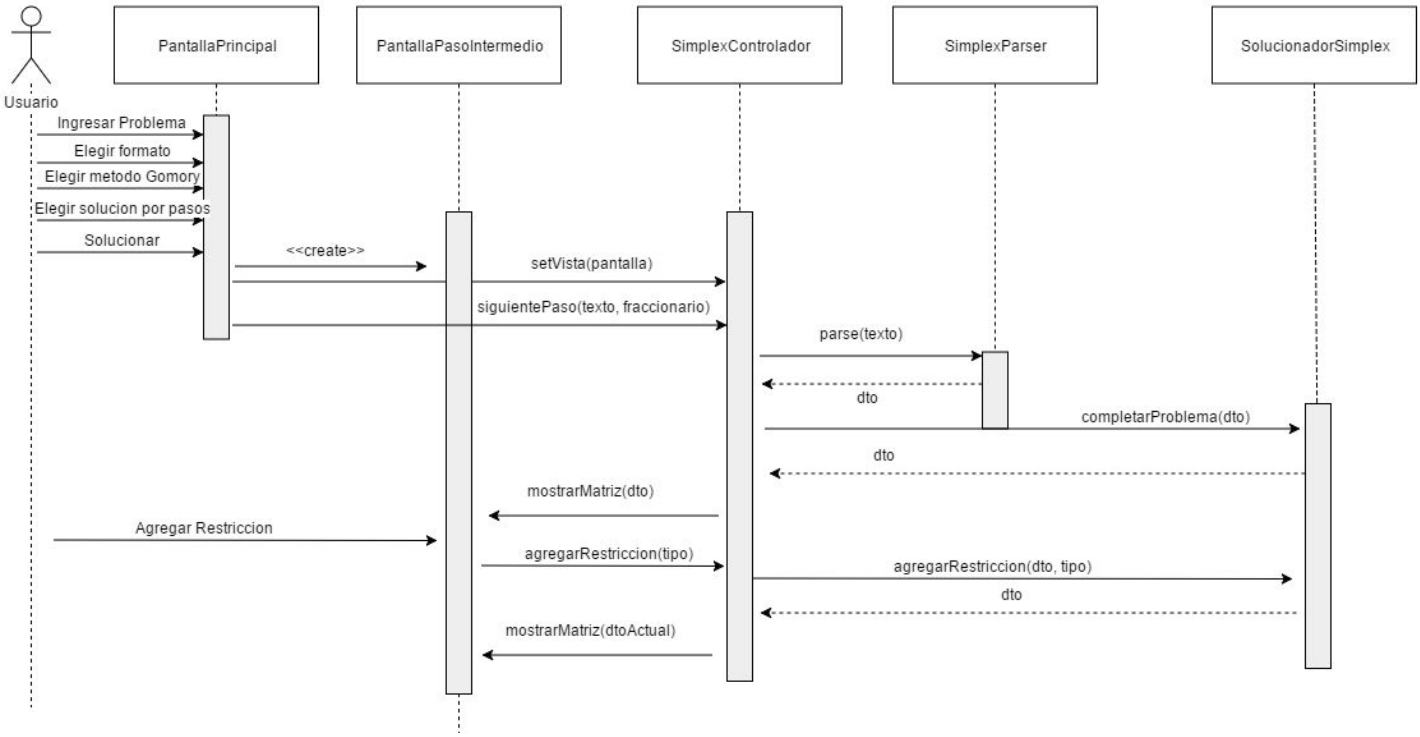
1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema de programación lineal que desea resolver en PantallaPrincipal.

4. El usuario escoge el método Simplex para solucionar el problema en PantallaPrincipal.
5. El usuario elige la opción de solución directa mostrada por PantallaPrincipal.
6. El usuario da clic en el botón solucionar.
7. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
8. PantallaPrincipal realiza inyección de dependencia a la clase ControladorBranchAndBound, y pasa una referencia a la recién creada PantallaPasoIntermedio para que ControladorBranchAndBound pueda actualizar la vista de manera acorde.
9. PantallaPrincipal indica a ControladorBranchAndBound que el usuario ha solicitado solucionar un problema entero paso por paso, y envía el texto que representa el problema junto con un valor booleano indicando el formato numérico de despliegue.
10. El hilo de ejecución es pasado a ControladorBranchAndBound. Esta clase es el centro de control de la lógica.
11. ControladorBranchAndBound toma el problema textual ingresado y mediante la clase SimplexParser abstrae la información presente en el problema y lo convierte a un Data Transfer Object (DTO) que será utilizado para pasar información durante el resto de la ejecución.
12. SimplexParser retorna el DTO a ControladorBranchAndBound.
13. Una vez con el problema textual representado por medio de un DTO, ControladorBranchAndBound envía esta información a GomoryBranchAndBound para encontrar la respuesta del problema.
14. SolucionadorBranchAndBoundcompleta el problema agregando las variables de holgura y artificiales necesarias, retornando el DTO completo a BranchAndBoundControlador.
15. Una vez que el DTO se encuentra completo, ControladorBranchAndBound le indica a SolucionadorBranchAndBound que muestre el siguiente paso del algoritmo, es decir, la siguiente iteración.
16. ControladorBranchAndBound actualiza la vista por medio del método mostrarMatriz y muestra el siguiente paso en PantallaPasoIntermedio.
17. PantallaPasoIntermedio muestra la matriz que representa el problema al usuario.
18. El usuario indica a PantallaPasointermedio que desea conocer el siguiente paso.

a. Mientras no se haya llegado a un óptimo y el problema sea factible y acotado, volver al punto 15.

19. Cuando se ha llegado a un óptimo, PantallaPasoIntermedio lo indica por medio de un mensaje y termina la ejecución.

6.10.2. Agregar restricciones a un problema de programación lineal.

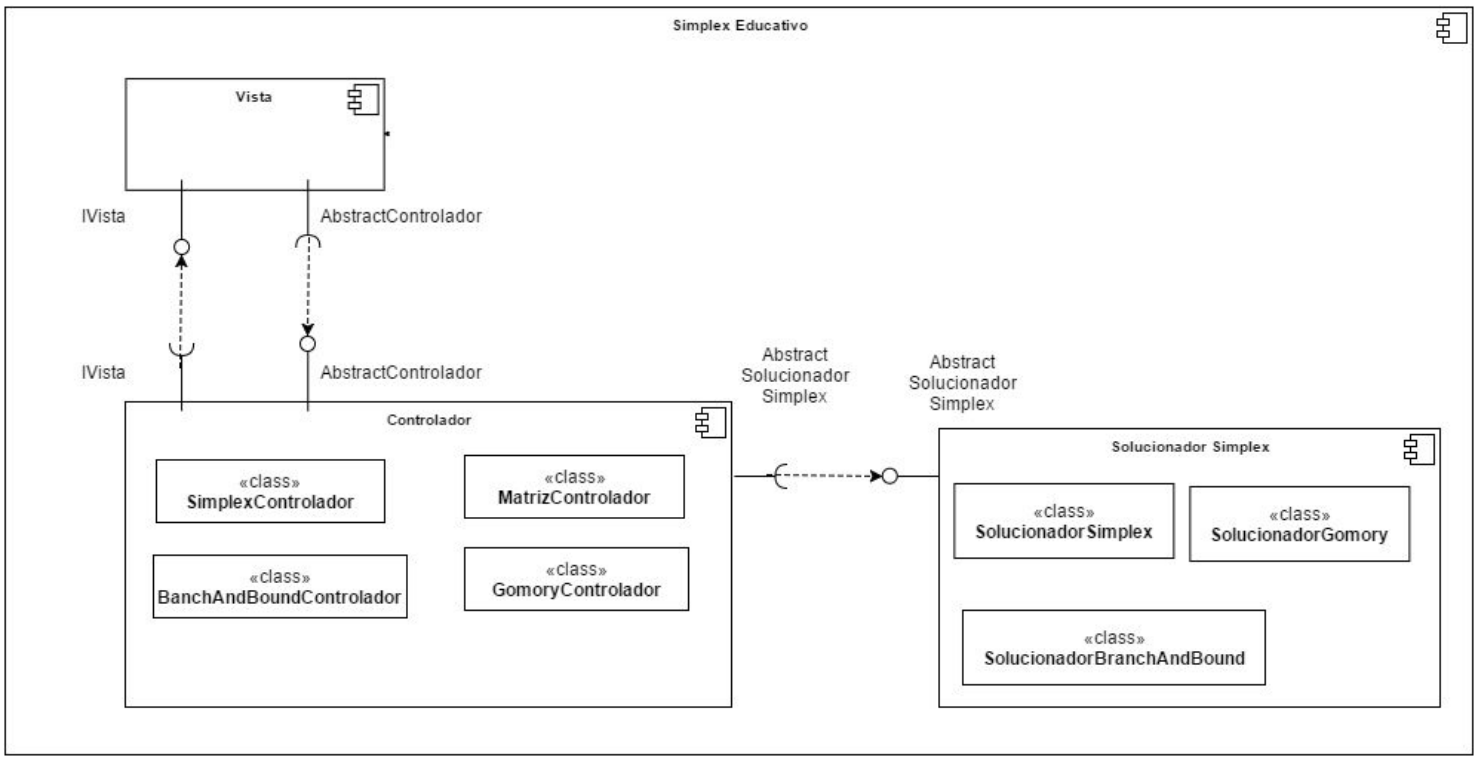


1. PantallaPrincipal es mostrada al usuario.
2. El usuario elige el formato numérico para el despliegue de los resultados: fraccionario o decimal.
3. El usuario ingresa el problema que desea resolver en PantallaPrincipal.
4. El usuario escoge el método que desee para solucionar el problema en PantallaPrincipal.
5. El usuario da clic en el botón solucionar.
6. La clase PantallaPrincipal se encarga de instanciar la nueva pantalla que será mostrada al usuario, PantallaPasoIntermedio.
7. PantallaPrincipal realiza inyección de dependencia al controlador respectivo, y pasa una referencia a la recién creada PantallaPasoIntermedio para que el controlador pueda actualizar la vista de manera acorde.
8. El controlador actualiza la vista con el la representación matricial respectiva.

9. El usuario navega con el mouse hasta la barra de menú y escoge la opción “Agregar restricción”.
10. El usuario hace clic con el mouse en el tipo de restricción que quiere agregar (los posibles son >=, <= ó =).
11. PantallaPasoIntermedio indica al controlador que el usuario ha agregado una nueva restricción, e invoca al método agregarRestriccion de la clase SimplexControlador con los parámetros necesarios (un DTO y el tipo de restricción agregada).
12. SimplexControlador utiliza la clase SolucionadorSImplex para agregar la restricción indicada por el usuario. Solucionador simplex retorna el DTO actualizado a SimplexControlador.
13. El controlador muestra la nueva matriz con la nueva restricción en PantallaPasoIntermedio, seguidamente actualiza el valor de los radios mostrados en pantalla, rellenando los radios mostrados de manera acorde.

7. Vista de Desarrollo

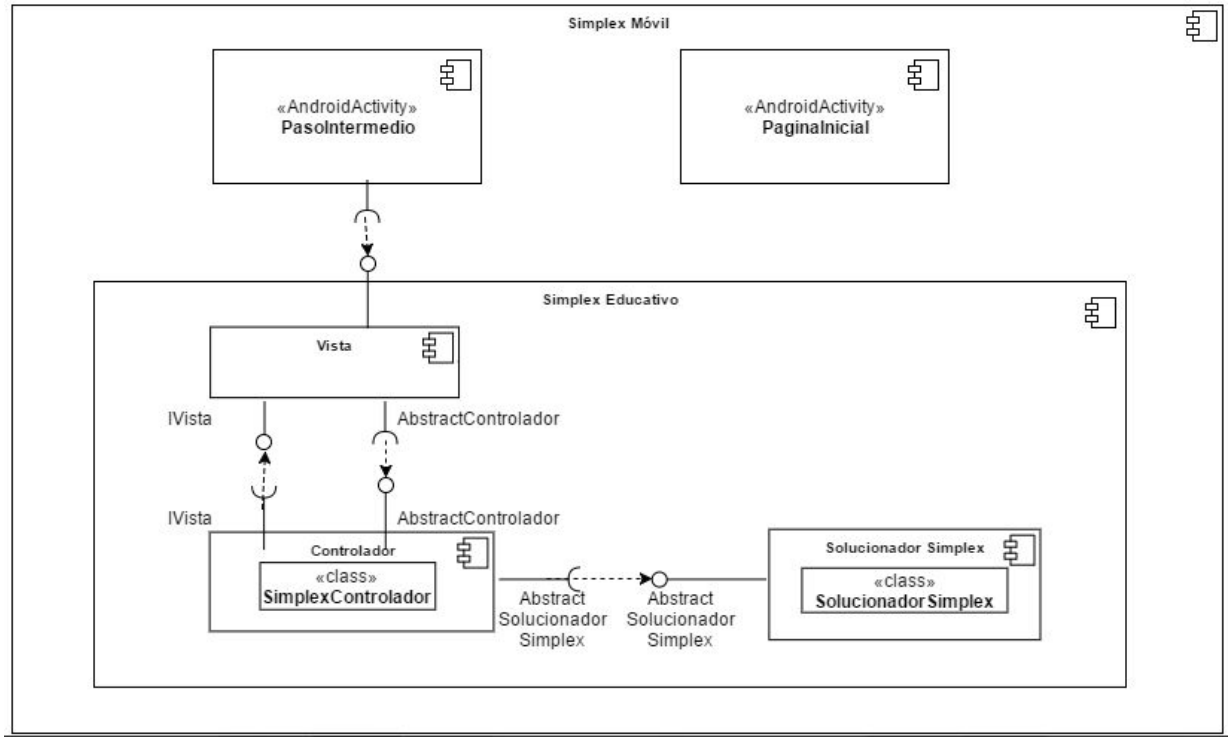
Dentro de esta sección se describirán, los componentes presentes dentro del sistema, donde se agrupan funcionalidades que posean un fin común dentro de los casos de uso o requerimientos funcionales. A continuación, se encuentra el diagrama de componentes donde se muestran los tres elementos identificados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



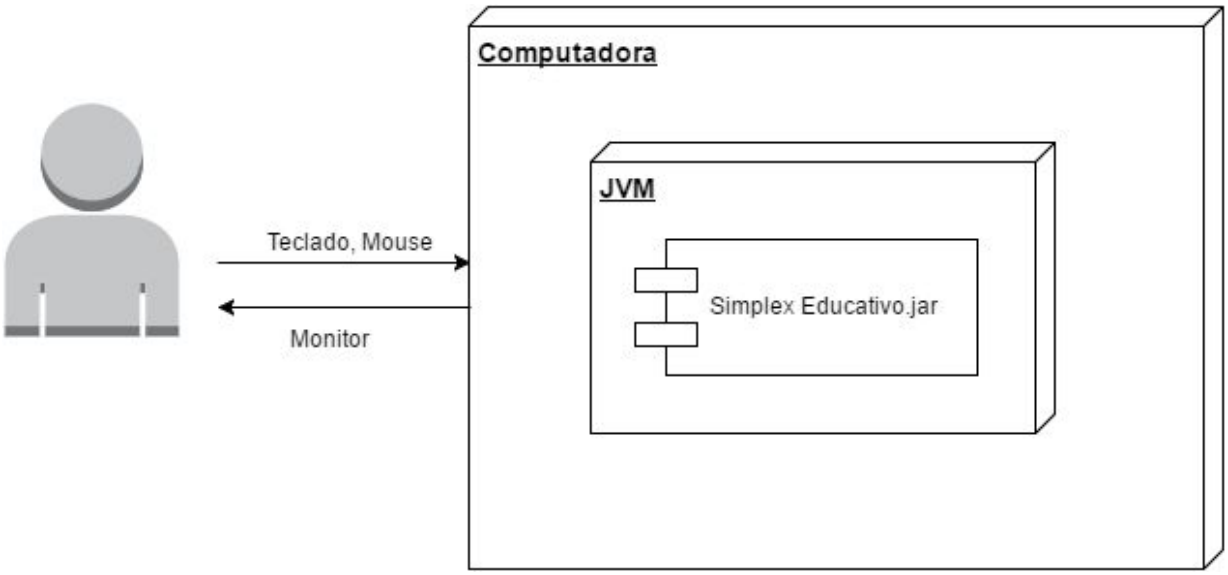
1. **Vista:** contiene la solución gráfica del programa, todos los artefactos pertenecientes a la interfaz gráfica que se le brinda al usuario final se almacenan dentro de este componente. Además se comunica con distintas interfaces del controlador donde se ejecuta validaciones y solicitudes de lógica 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 por medio del controlador respectivo, para luego procesarla y validar el resultado, retornando una solución veraz al caso que se solicitó.

Versión Móvil

Para la versión móvil del sistema Simplex Educativo se aprovechó la portabilidad del lenguaje de programación Java y su integración con el ambiente de desarrollo Android para reutilizar el trabajo hecho durante la iteración 1 y 2 y utilizar la solución ya desarrollada para el ambiente móvil. El sistema operativo Android es capaz de ejecutar código escrito en Java, por lo tanto, bastó con importar una librería al proyecto móvil para tener acceso a todas las funcionalidad ya desarrolladas anteriormente. Por lo tanto, solamente fue necesario desarrollar la interfaz gráfica de la aplicación para llamar a los servicios provistos por la librería.

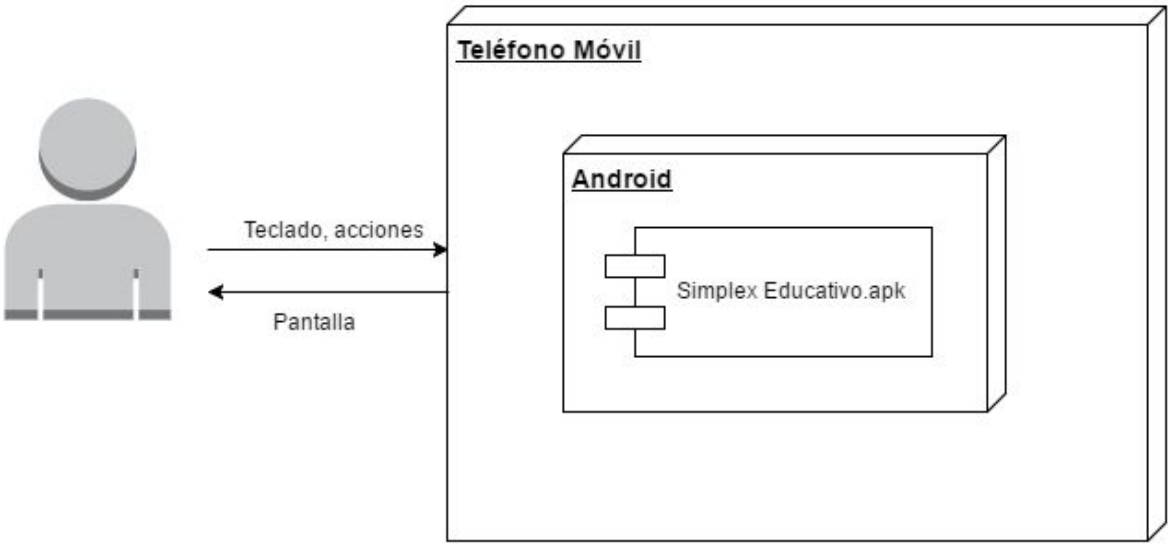


8. Vista de Despliegue



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.

Versión Móvil



De la misma manera que la versión de escritorio, es constituido por un único nodo debido a que no hay interacción con otros sistemas o subsistemas. El usuario interactúa con la aplicación por medio de su teléfono móvil, con acciones mediante sus dedos y el teclado del teléfono. Por su parte, la aplicación interactúa con el usuario por medio de la pantalla, mostrando la información de cada iteración del algoritmo en caso de la solución por pasos o directamente la solución buscada.

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/Java/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/Java/modelo/parser/Parser.cup y puede ser compilada mediante el siguiente comando:

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