UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN (DCC)

Documento de Diseño

JHawanetFramework: Herramienta para la optimización de redes de distribución de agua potable

Fecha: 2 de junio del 2017

Versión: 1.0

Equipo de Desarrollo:

Nombre	Rol	Contacto
Gabriel Sanhueza Fuentes	Administrador, Analista, Diseñador, Implementador y Tester.	gsanhueza15@alumnos.utalca.cl

Contraparte:

Nombre	Rol	Contacto
Jimmy H. Gutiérrez-Bahamondes	Cliente/Profesor guía	
Daniel Mora-Meliá	Cliente/Profesor co-guía	

Historia del Documento

Versión	Fecha	Razón del Cambio	Autor(es)
0.1	07/09/2019	Primer borrador	
0.2	30/09/2019	Rellenar_arquitectura_logica	
0.3	05/09/2019	Agregar diagramas de clases	
0.4	12/12/2019	Agregado diagrama de secuencia y módulos	
0.5	13/12/2019	Agregado detalle de anotaciones	

Tabla de contenido

Historia del Documento 2
1. Introducción 4
1.1 Propósito del Sistema 4
1.2 Alcance del Proyecto 4
1.3 Definiciones, siglas y abreviaturas 4
1.4 Referencias 5
1.5 Descripción general 5
2. Diseño Arquitectónico 6
2.1. Arquitectura Física 6
2.2. Arquitectura 6
Componente Metaheurísticas (Metaheuristics): 7
Componente Hidráulico (EPANET): 7
3. Diseño Detallado 8
3.1. Diseño detallado de módulos 8
3.2. Diseño de estructura de sistema 9
3.3. Operación del Sistema 11
4. Diseño de Interfaz 12
4.1 Diseño de interfaz de usuario 12
4.2 Interfaz de inicio. 12
4.3 Interfaz de configuración del problema 12
4.4 Interfaz de ejecución de algoritmos 13
4.5 Interfaz de gráficos 13
4.6 Interfaz de resultados 14
5 Detalles de implementación 16
5.1 Uso de anotaciones de java (Java Annotation) y reflexión (Reflection) 16
6 Matriz de trazado 17

1. Introducción

1.1 Propósito del Sistema

El proyecto consiste en el desarrollo de un sistema que permita simular y buscar soluciones a problemas presentes en las redes de distribución de agua potable haciendo uso de algoritmos metaheurísticos. Además, este sistema será desarrollado de tal forma que sea escalable con el fin de que otros desarrolles o investigadores sean capaces de extender su funcionalidad fácilmente agregando nuevos algoritmos metaheurísticos y problemas a tratar en el contexto de la distribución de redes de agua potable.

1.2 Alcance del Proyecto

Al final del periodo de desarrollo la herramienta contara con las siguientes prestaciones.

- El sistema permitirá la carga y la visualización de la red gráficamente.
- El sistema inicialmente solo resolverá dos clases de problemas de optimización, uno monoobjetivo y el otro multiobjetivo. El problema monoobjetivo será el de los costos de inversión. En cuanto al problema multiobjetivo, este será el de los costos energéticos y el número de encendidos y apagado de las bombas.
- El sistema inicialmente únicamente contara con dos algoritmos implementados los cuales serán el algoritmo genético y NSGA-II. El algoritmo genético será el usado para tratar el problema monoobjetivo, mientras que NSGA-II será aplicado al multiobjetivo.

Este proyecto no contempla la creación de la red por lo que estas deberán ser ingresadas como entradas al programa. La creación de las redes puede realizarse a través de EPANET. Además, esta herramienta únicamente podrá ser ocupada en equipos cuyo sistema operativo sea Windows debido a que se realizan llamadas a librerías nativas.

1.3 Definiciones, siglas y abreviaturas

- *Genetic Algorithm* (GA): Estrategia de búsqueda de soluciones basada en la teoría de la evolución de Darwin. Para realizar esto, el algoritmo parte desde un conjunto de soluciones denominada población he iterativamente, lleva a cabo un proceso de reproducción, generando nuevas soluciones [1].
- *Non-dominated Sorting Genetic Algorithm* (NSGA-II): Algoritmo que utiliza el cruzamiento, mutación y reproducción para encontrar un conjunto de soluciones optimas a problemas que cuentan con más de un objetivo [2].
- Metaheuristicas: Algoritmos que permiten resolver un amplio rango de problemas de optimización empleando técnicas con algún grado de aleatoriedad para encontrar soluciones a un problema. Estos algoritmos no garantizan que la solución encontrada sea la óptima, pero permiten obtener generalmente aproximaciones a esta [3]–[5].
- *Java Reflection*: Característica de java que permite que un programa se auto examine. Esta característica está disponible a través de la *Java Reflection API,* la cual cuenta con métodos para obtener los *meta-object* de las clases, métodos, constructores, campos o parámetros. Esta API también permite crear nuevos objetos cuyo tipo era desconocido al momento de compilar el programa [6].

- *Java Annotation*: Característica de java para agregar metadatos a elementos de java (clases, métodos, parámetros, etc.) [7]. Las anotaciones no tienen efecto directo sobre el código, pero cuando son usadas junto con otras herramientas pueden llegar a ser muy útiles. Estas herramientas pueden analizar estas anotaciones y realizar acciones en base a estas, por ejemplo, generar archivos adicionales como clases de java, archivos xml, entre otras; ser analizadas durante la ejecución del programa vía *Java Reflection*, para crear objetos cuyo tipo no conocemos en tiempo de compilación; etc.

1.4 Referencias

[1] D. Heiss-Czedik, "An Introduction to Genetic Algorithms.," *Artif. Life*, vol. 3, no. 1, pp. 63–65, 1997.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, 2002.

[3] X. S. Yang, "Metaheuristic Optimization," vol. 6, no. 2011, 2015.

[4] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Inf. Sci. (Ny).*, vol. 237, pp. 82–117, 2013.

[5] S. Luke, *Essentials of Metaheuristics*, Second. Lulu, 2013.

[6] M. Braux and J. Noyé, "Towards partially evaluating reflection in Java," *ACM SIGPLAN Not.*, vol. 34, no. 11, pp. 2–11, 1999.

[7] H. Rocha and M. T. Valente, "How annotations are used in Java: An empirical study," *SEKE 2011 - Proc. 23rd Int. Conf. Softw. Eng. Knowl. Eng.*, no. June 2014, pp. 426–431, 2011.

[8] A. Syromiatnikov and D. Weyns, "A journey through the land of model-view-design patterns," *Proc. - Work. IEEE/IFIP Conf. Softw. Archit. 2014, WICSA 2014*, pp. 21–30, 2014.

[9] A. J. Nebro, J. J. Durillo, and M. Vergne, "Redesigning the jMetal multi-objective optimization framework," in *GECCO 2015 - Companion Publication of the 2015 Genetic and Evolutionary Computation Conference*, 2015.

1.5 Descripción general

El proyecto será un medio que permitirá la realización de simulaciones hidráulicas aplicando metaheurísticas con el fin de buscar soluciones a problemas comunes en redes de distribución de agua potable.

Inicialmente se incorporará la presencia de dos metaheurísticas las cuales serán el Algoritmo Genético y el algoritmo NSGA-II. Cada uno de estos algoritmos estará enfocado en la búsqueda de soluciones para un problema determinado. El algoritmo GA estará enfocado en la búsqueda de soluciones para el objetivo de costos de inversión desde el enfoque monoobjetivo. En cambio, el algoritmo NSGA-II buscara soluciones para el problema multiobjetivo, cuyos objetivos serán el costo de operación y el régimen de bombeo (Pumping Scheduling).

2. Diseño Arquitectónico

2.1.Arquitectura Física

La arquitectura física del software a desarrollar consistirá en la arquitectura monolítica. Puesto que la herramienta a desarrollar es un programa de escritorio cuya persistencia de datos se realiza en el mismo equipo del usuario que está ejecutando la aplicación. En la Ilustración 1 se puede ver una imagen que representa la arquitectura.

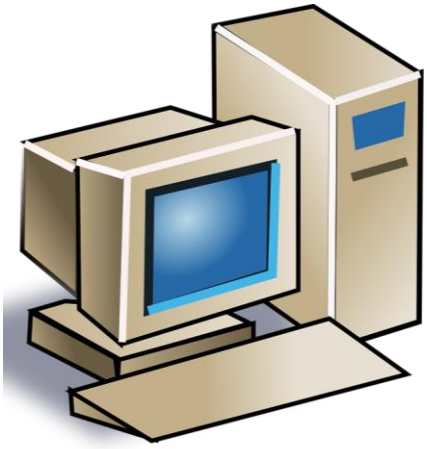

Ilustración 1, diagrama de arquitectura física.

2.2.Arquitectura Lógica

La arquitectura lógica que se utilizara corresponde al patrón Modelo-Vista-Controlador [8]. La elección de este patrón se debe a que permite la separación de la lógica de negocio y la vista presentada al usuario logrando de esta manera una aplicación altamente mantenible y con una mejor escalabilidad. El diagrama que representa la arquitectura usada para el desarrollo se puede ver en la Ilustración 2.

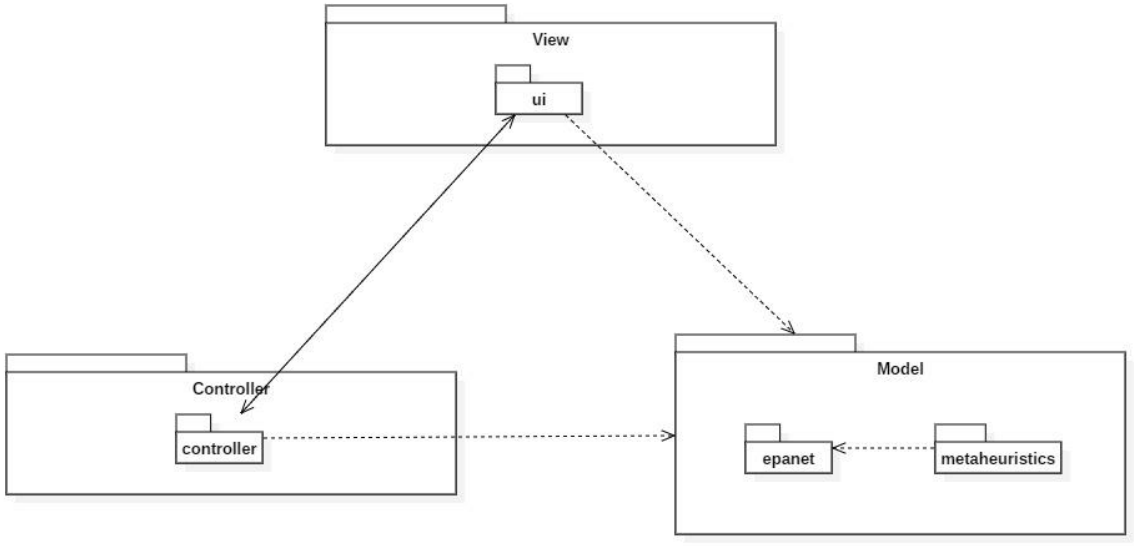

Ilustración 2, diagrama de arquitectura lógica.

En el diagrama presentado en la Ilustración 2 se presencia la división de la aplicación en tres capas. La capa Vista se encarga de la interacción con el usuario y de la interfaz de usuario. La capa Controlador se encarga de responder a los eventos de los usuarios y solicitar información a la capa de modelo. La capa Modelo contiene la información y la lógica fundamental de la aplicación en un formato adecuado para interactuar con las demás capas. Esta capa, también se encarga de la ejecución de los algoritmos y la interacción con la librería nativa EpanetToolkit.

La capa modelo se divide principalmente en dos componentes. Estos componentes son:

Componente Metaheurísticas (Metaheuristics):

Modulo que contiene los algoritmos, operadores, los tipos de solución permitidos y los problemas que serán abarcados durante el desarrollo del proyecto.

Componente Hidráulico (EPANET):

El módulo hidráulico posee las clases necesarias para cargar los archivos de red (inp) y generar una representación de ellos a través de una serie de clases. Este módulo también se encarga de guardar la representación de una red ya modificada en un archivo inp para que pueda ser usado por el programa EPANET. Las simulaciones hidráulicas serán realizadas usando la librería epajava. Esta librería realiza las llamadas nativas a la EpanetToolkit. La librería epajava puede ser descargada en el repositorio de git ubicado en https://github.com/jhawanet/epajava .

3. Diseño Detallado

3.1. Diseño detallado de módulos

Los componentes que forman la aplicación son el componente de la GUI, el de los Controlladores, el componente de Epanet y el componente Metaheuristico. La relación entre los componentes puede ser apreciada en la Ilustración 3.

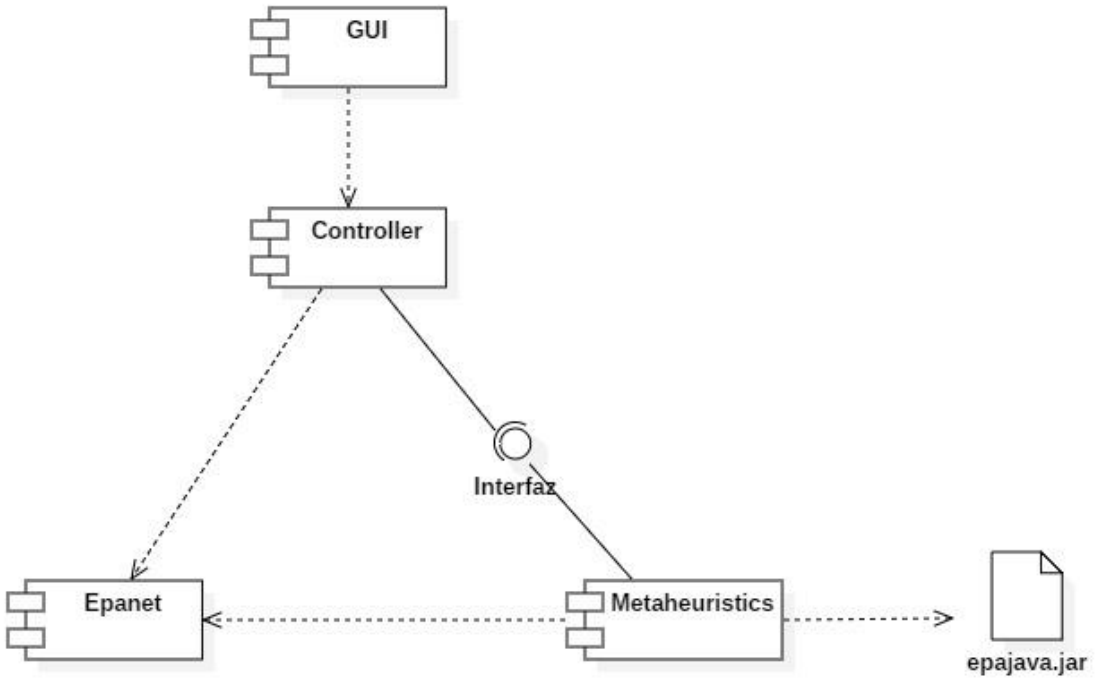

Ilustración 3, Diagrama de componentes

A continuación, se describe cada uno de los componentes presentados en la Ilustración 3:

GUI (Graphics User Interface) Este componente este encargado de presentar todas las vistas. Entre las vistas se encuentra la ventana principal de la aplicación, la ventana de configuración de para un problema, la ventana de ejecución de algoritmos y la ventana de resultados.

Controller El controlador se encarga de manejar los eventos generados por la GUI. Generalmente la relación es uno a uno, es decir, por cada interfaz de usuario hay un controlador. Debido a que dentro de la interfaz de usuario esta formada por varios componentes, se da el caso en que

	cada uno de estos componentes puede tener su propio controlador.
Epanet	Este componente esta encargada de la lectura y la escritura de archivos inp. Este componente cuenta también con clases que representan una red cargada desde un inp. Estas clases permitirán editar, durante la ejecución del programa, algunas configuraciones de la red con el fin de crear un nuevo archivo de descripción de la red.
Metaheuristics	Este componente contiene los algoritmos metaheurísticos, así como los problemas y los operadores que pueden ser ocupados por los algoritmos.
epajava.jar	Esta es una librería para la simulación de las redes de agua potable. Esta librería permite realizar llamadas nativas a la librería de Epanet. Estas llamadas nativas se hacen a través de la librería JNA (Java Native Access). Para ocupar la librería, esta requiere que se indique el archivo de descripción de la red (Archivo inp).

3.2.Diseño de estructura de sistema

En la Ilustración 4 se muestra un diagrama general de los componentes, sus clases mas importantes y su interacción.

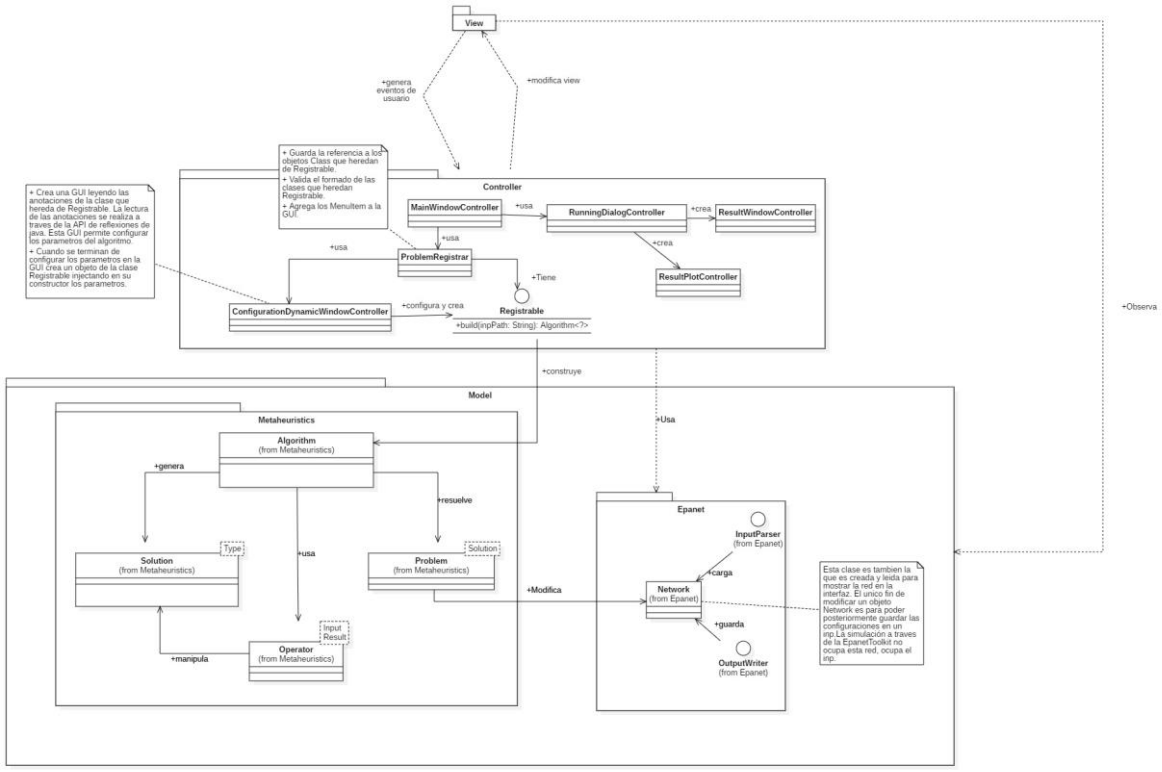

Ilustración 4, diagrama de clases general del sistema

La Ilustración 5 corresponde a un diagrama de clases mas detallado del componente Epanet.

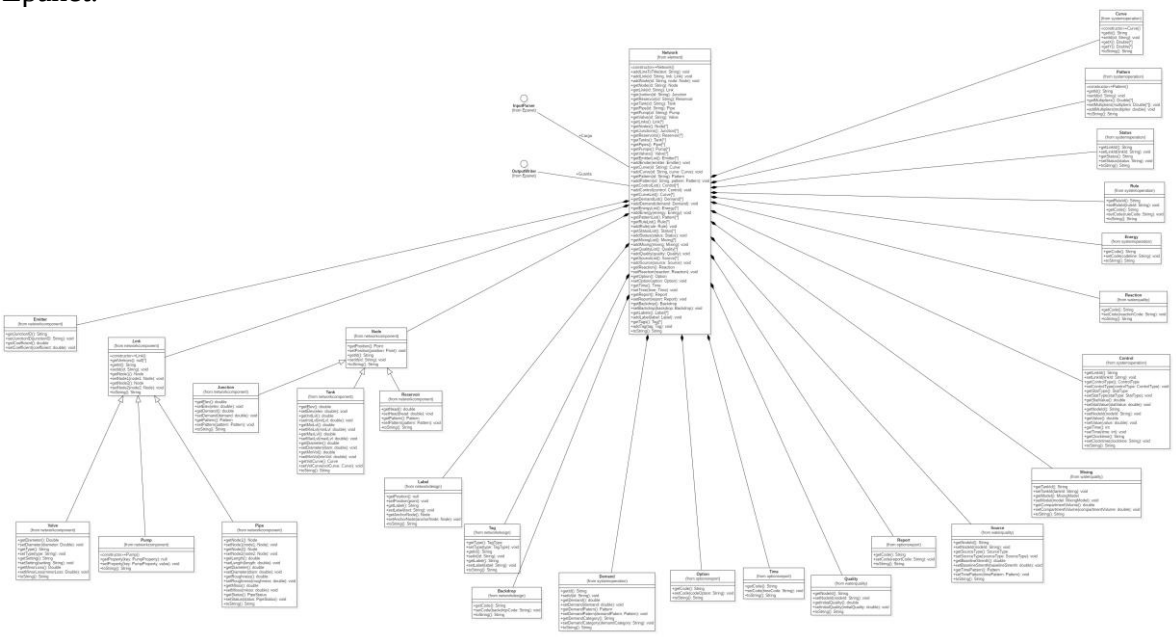

Ilustración 5, diagrama de clases del componente Epanet.

En la Ilustración 6, se presenta el diagrama de clases del componente metaheurístico. Este fue tomado de [9] y adaptado para ser usado por nuestra aplicación.

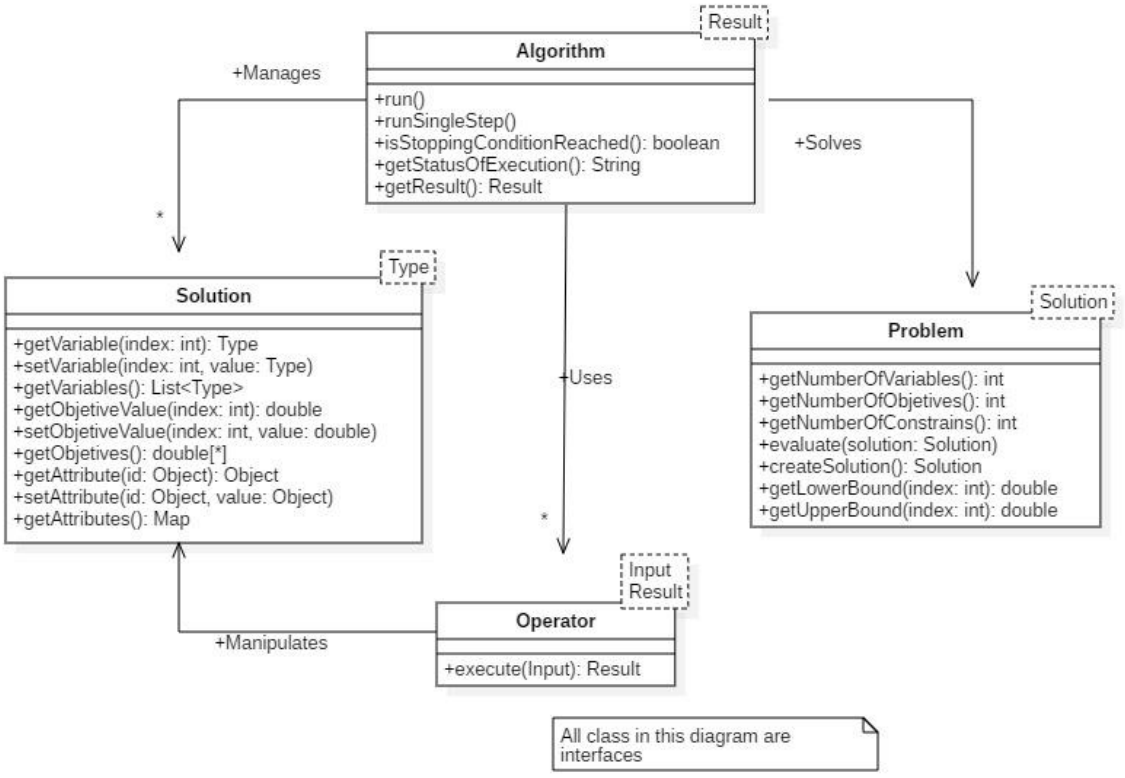

Ilustración 6, diagrama de clases del componente metaheuristics.

3.3.Operación del Sistema

En la ilustración 7 se puede ver a mas detalle como interacciona el usuario con la aplicación.

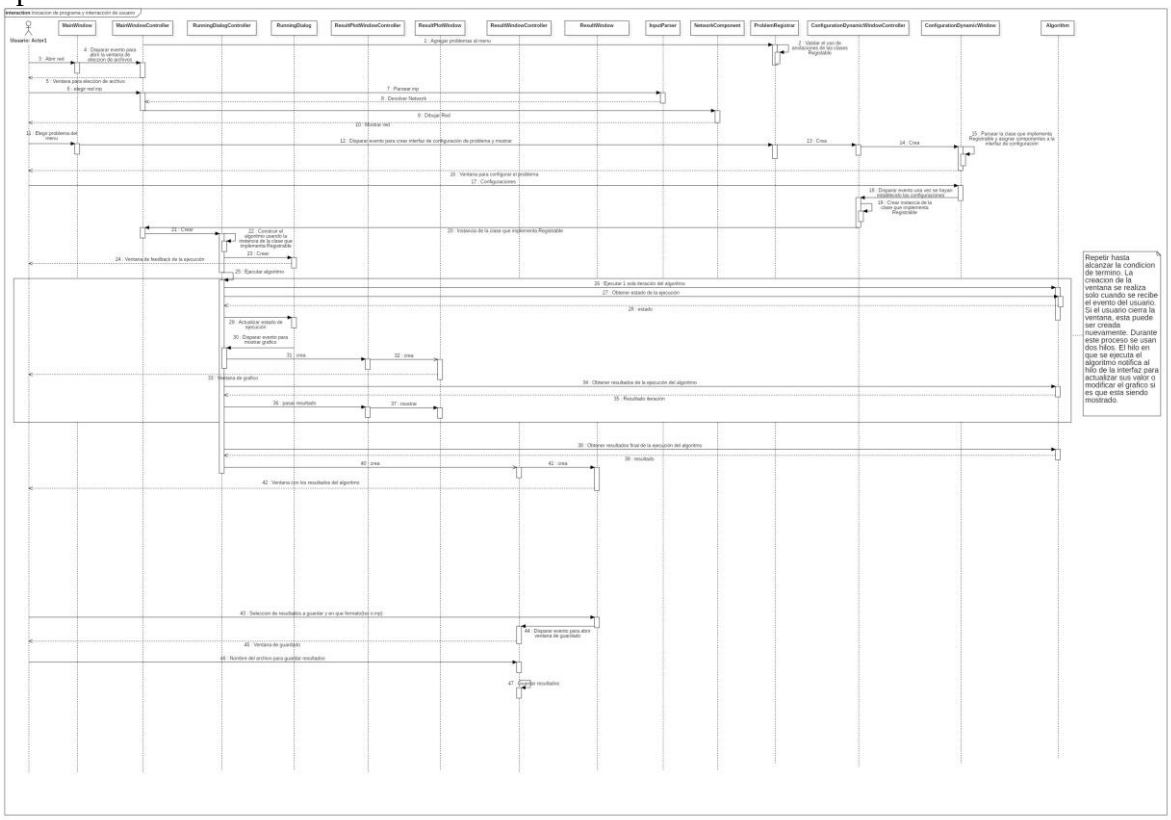

Ilustración 7, diagrama de secuencia para la operación del sistema.

4. Diseño de Interfaz

4.1 Diseño de interfaz de usuario

4.2 Interfaz de inicio.

La interfaz de inicio se divide en tres secciones, el menú, el visualizador de red y un apartado para ver los elementos de la red. Para cargar una red debe ir a *File* > *Open*. Para resolver un problema debe ir al menú *Problems* y elegir el problema a resolver. Para ver mas detalles de un componente de la red, entonces pulse dos veces en componente de la red en el apartado de elementos. Esta interfaz se muestra en la Ilustración 8.

Ilustración 8, esquema de interfaz de inicio del sistema.

4.3 Interfaz de configuración del problema

Esta interfaz es mostrada solo cuando el problema tiene parámetros que configurar.

Cuando se seleccione un operador que en su constructor por defecto (indicado por la anotación *@DefaultConstructor*) reciba parámetros se habilitara el botón de configuración.

Ilustración 9, esquema de interfaz de configuración del problema

4.4 Interfaz de ejecución de algoritmos

Esta interfaz es mostrada mientras se lleva a cabo la ejecución del algoritmo. Esta muestra un *TextField* cuyo texto es obtenido a través del método *getStatusOfExecution* de la clase *Algorithm.* Puede ver como luce esta interfaz en la Ilustración 10.

Si se presiona el botón cancelar, entonces la ejecución del algoritmo será detenida. Por otra parte.
El botón "*Show chart*" estará disponible para problema de hasta dos objetivos.

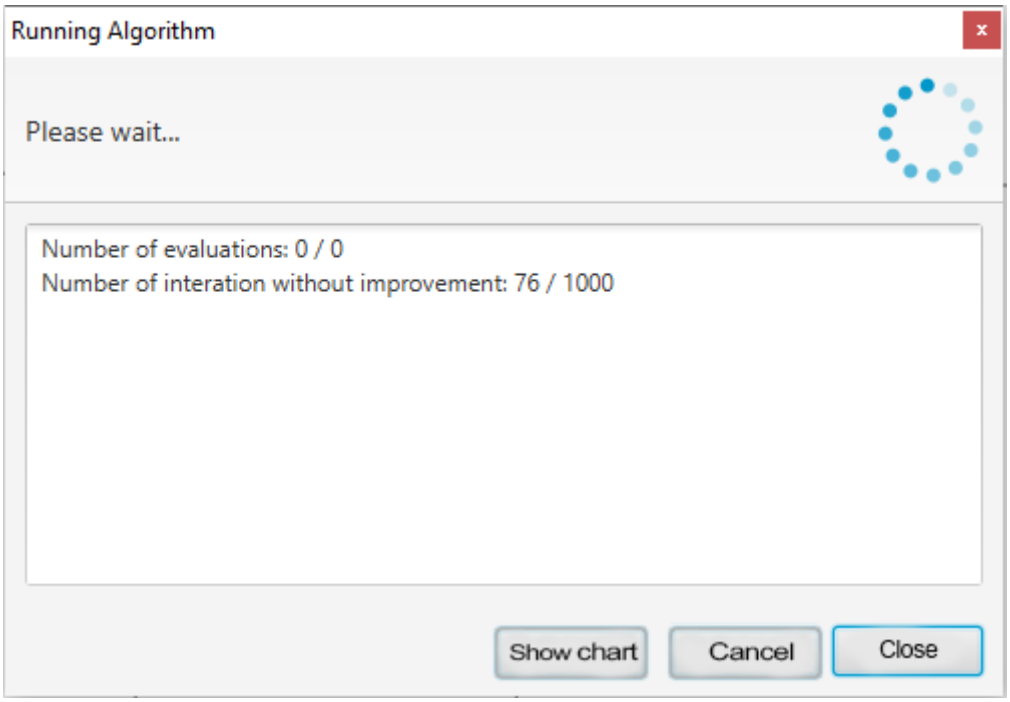

Ilustración 10, esquema de interfaz de ejecución de algoritmo.

4.5 Interfaz de gráficos

Esta interfaz será mostrada cuando el botón "*Show chart"* sea presionado. Esta interfaz contara con un grafico de dos ejes. Si el problema es de un objetivo, entonces el eje vertical corresponderá al valor del objetivo después de realizar la evaluación. Mientras que el eje horizontal corresponderá al numero de iteraciones. Si el problema tiene dos objetivos, el eje vertical corresponderá al primer objetivo y el eje horizontal corresponderá al segundo objetivo. Esta interfaz es mostrada en la Ilustración 11.

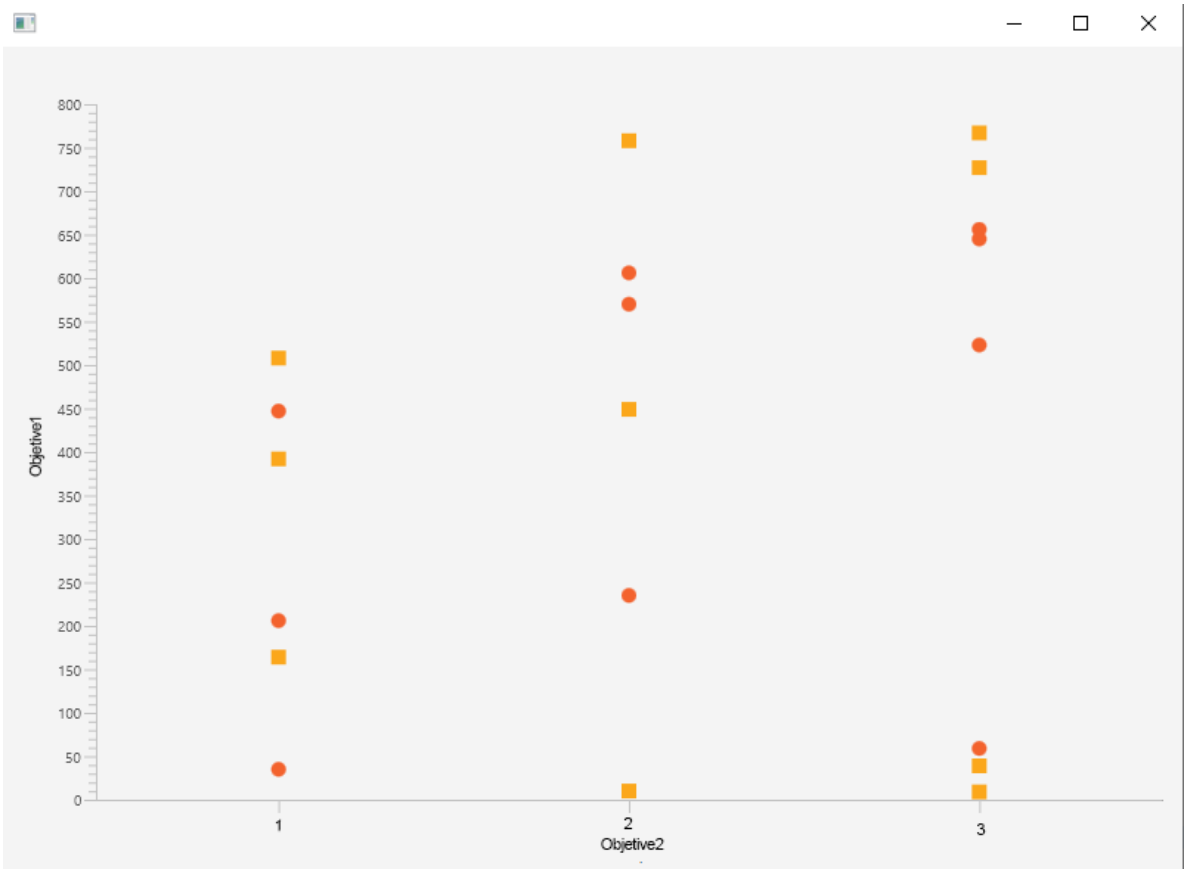

Ilustración 11, esquema de interfaz de gráficos.

4.6 Interfaz de resultados

La interfaz de resultados será mostrada cuando la ejecución del algoritmo haya terminado exitosamente. Esta interfaz cuenta con dos botones como se muestra en la Ilustración 12.

El botón "*Save selected ítems as inp"* crea un inp para cada una de las soluciones seleccionadas. Para esto se envía una copia del objeto Network abierto y la solución al método XXX del problema, este método sustituye en el objeto Network los valores correspondientes indicados en la solución y devuelve nuevamente el objeto Network para poder guardarlo usando un objeto que implementa la interfaz *OutputWriter*.

El botón "*Save Table"* guarda todas las soluciones, en dos archivos separados. Uno de estos archivos guarda solamente las variables de decisión, y el otro guarda los valores de los objetivos. El nombre de estos corresponde al dado a través del FileChooser que es mostrado al presionar el botón. A este nombre se le agrega el sufijo -FUN, para el archivo con los valores de los objetivos; y -VAR, para el archivo con los valores de las variables de decisión.

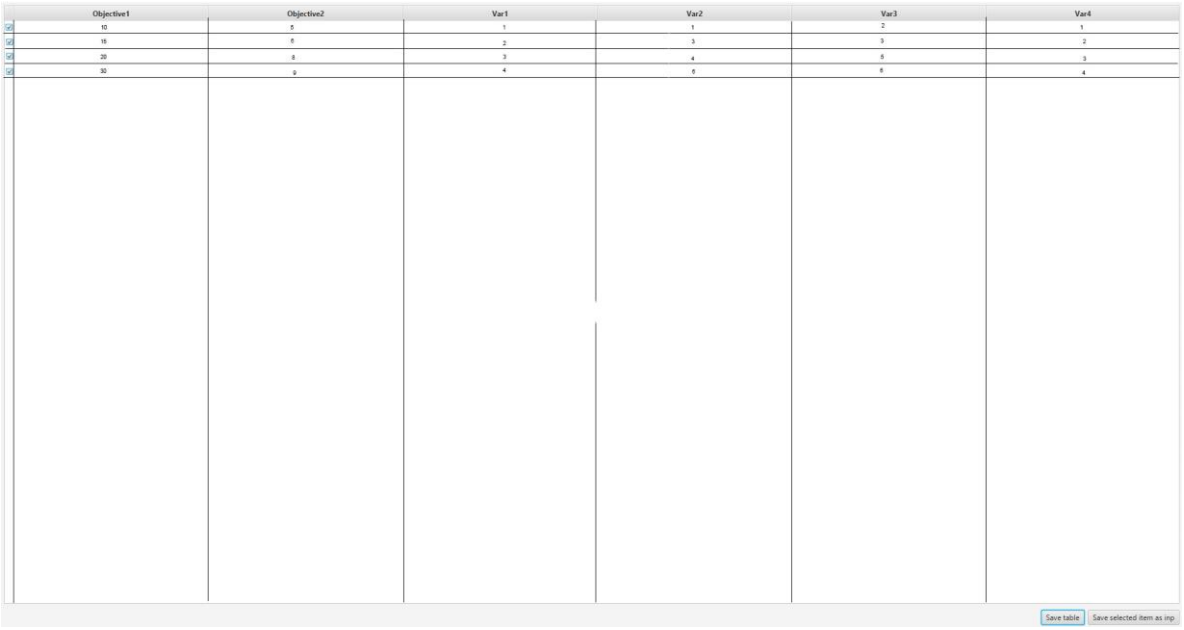

Ilustración 12, esquema de interfaz de resultados.

5 Detalles de implementación

Una necesidad del sistema es poder añadir nuevos algoritmos, operadores y problemas. Sin embargo, para hacer uso de estos desde la interfaz de usuario, sería necesario que el implementador modifique manualmente la interfaz. Esto llevaría una carga extra al implementador al tener que aprender la tecnología necesaria que fue usada para crear la interfaz, la cual para este sistema corresponde a JavaFX. Debido a estas razones, para facilitar el trabajo del implementador se tomo como alternativa usar dos tecnologías del lenguaje de Java, las cuales son *Java Reflection* y *Java Annotation*.

El uso que se hace por parte de ellas en el sistema es:

Java Reflection	Examina las clases durante la ejecución del programa con el fin de construir una interfaz de usuario que permita configurar los valores que serán necesarios al momento de crear un objeto de dicha clase.
Java Annotation	Agrega metadatos a los elementos del programa, en este caso a los constructores, que serán leídos a través de la *Java Reflection API*. Estos metadatos contienen información del constructor como el nombre de los parámetros que recibe y en el caso de que el parámetro sea un objeto, las alternativas de las clases para crear dicho objeto.

Haciendo uso de estas dos tecnologías, se puede reducir esta preocupación, puesto que, estableciendo y siguiendo una convención se puede crear dinámicamente una la GUI para instanciar nuevos objetos sin conocer su tipo previamente en tiempo de compilación. La convención anteriormente mencionada, consiste en implementar una interfaz, en la cual el constructor indique los parámetros que requiere, los cuales deben estar en un orden determinado, para crear y configurar el algoritmo cuando este sea solicitado a través del método declarado por la interfaz. El nombre de dicha interfaz es Registrable.

5.1 Uso de *Java Annotation* y *Java Reflection*

Las anotaciones presentes en el sistema, su finalidad y donde deberían ser usadas se define a continuación:

5.1.1 Anotaciones para los operadores:

@DefaultConstructor

Indica el constructor que debe ser usado al momento de crear una instancia del operador. Esta anotación recibe un arreglo de *String* con el nombre de cada operador. El arreglo debe tener la misma cantidad de argumentos que los parámetros del constructor como se muestra en la ilustración 13 y 14.

```
@DefaultConstructor({"constant"})
public UniformSelection(double constant)
```

Ilustración 13, constructor de un solo parametro

```
@DefaultConstructor({"MutationProbability", "DistributionIndex"})
public IntegerPolynomialMutation(double mutationProbability, double distributionIndex)
```

Ilustración 14, constructor de múltiples parámetros

Esta anotación solo puede ser usada en un constructor por clase. Usar esta anotación en mas de un constructor lanzara una excepción en tiempo de ejecución. Adicionalmente, el constructor que use esta anotación solo puede tener parámetros de tipo *int* o *double*.

La interfaz creada para cada anotación se puede ver en la ilustración 15 y 16.

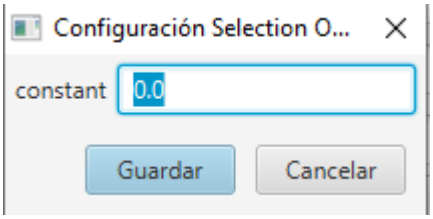

Ilustración 15, interfaz para configurar el operador *UniformSelection*

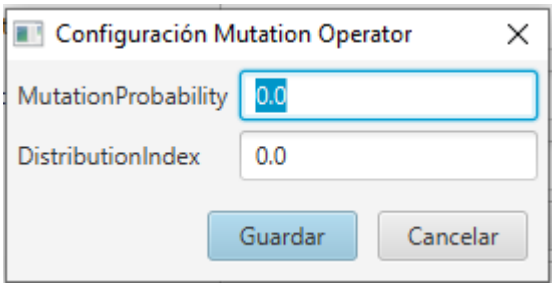

Ilustración 16, interfaz para configurar el operador *IntegerPolynomialMutation*

La anotación puede tener un arreglo vacío, lo cual indica que el constructor no recibirá parámetros.

5.1.2 Anotaciones para los objetos que heredan la interfaz Registrable:

@NewProblem

Esta anotación permite indicar el nombre del problema que será mostrado en la interfaz gráfica. Puedes ver el uso de esta anotación en la Ilustración 17.

```
@NewProblem(displayName = "Cost problem")
@Parameters(operators = {
        @OperatorInput(displayName = "Selection Operator", value = {
                @OperatorOption(displayName = "Uniform Selection", value = UniformSelection.class) }),
        @OperatorInput(displayName = "Crossover Operator", value = {
                @OperatorOption(displayName = "Integer SBX Crossover", value = IntegerSBXCrossover.class),
                @OperatorOption(displayName = "Integer Single Point Crossover", value = IntegerSinglePointCrossover.class) }), //
        @OperatorInput(displayName = "Mutation Operator", value = {
                @OperatorOption(displayName = "Integer Polynomial Mutation", value = IntegerPolynomialMutation.class),
                @OperatorOption(displayName = "Integer Range Random Mutation", value = IntegerRangeRandomMutation.class),
                @OperatorOption(displayName = "Integer Simple Random Mutation", value = IntegerSimpleRandomMutation.class) }) }, //
        files = { @FileInput(displayName = "Gama") }, //
        numbers = { @NumberInput(displayName = "Number of iteration without improvement"),
                @NumberInput(displayName = "Max number of evaluation") })
public InversionCostRegister(Object selectionOperator, Object crossoverOperator, Object mutationOperator, File gama,
        int numberWithoutImprovement, int maxEvaluations)
```

Ilustración 17, constructor de clase que hereda de registrable y sus metadatos para cada parámetro.

El nombre dado en el elemento *displayName*, es el nombre visible en el menú de la aplicación, como se ve en la Ilustración 18.

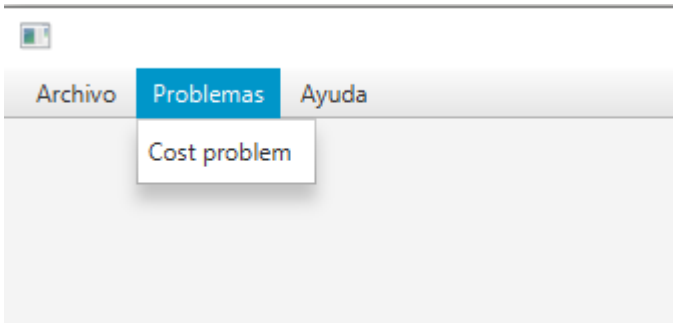

Ilustración 18, menú de problemas

Esta anotación solo puede estar en un constructor, en caso de esta anotación no este presente un error en tiempo de ejecución será lanzado.

@Parameters

Esta anotación permite agregar información acerca de los parámetros recibidos por el constructor. Cuando el constructor tiene esta anotación, por convención, está obligado a declarar los parámetros en un orden determinado en base a tu tipo. Este orden es el siguiente:

1. *Object*
2. *File*
3. *int* o *double*

Si el constructor no declara los parámetros en ese orden un error en tiempo de ejecución será lanzado.

Dentro de esta anotación existen varios elementos. Estos elementos son:

- *operators:* Arreglo que recibe anotaciones del tipo *OperatorInput*.
- *files*: Arreglo que recibe anotaciones del tipo *FileInput*.

- *numbers:* Arreglo que recibe anotaciones del tipo *NumberInput.*
- *numbersToggle:* Arreglo que recibe anotaciones del tipo *NumberToggleInput*.

El valor por defecto para todos los elementos mencionados anteriormente es un arreglo vacío ({}).

No usar la anotación *@Parameters* tiene el mismo efecto que usar la anotación pero con sus valores por defecto.

Puede ver el uso de esta anotación en la ilustración 17.

@OperatorInput

Esta anotación agrega mas información a uno de los parámetros del constructor.

Dentro de esta anotación existen varios elementos. Estos elementos son:

- *displayName:* Nombre de categoría para los operadores.
- *value*: Arreglo que recibe anotaciones del tipo *OperatorInput*.

En la ventana de configuración de problema, estos parámetros son vistos con un ComboBox como muestra la ilustración 19.

Ilustración 19, componente *ComboBox* para configurar los operadores.

Las alternativas disponibles dentro del ComboBox están dadas por aquellas indicadas en el elemento *value* de este operador. Como se muestra en la ilustración 17, la única alternativa para el *Selection Operator* es el operador UniformSelection apreciado en la ilustración 20.

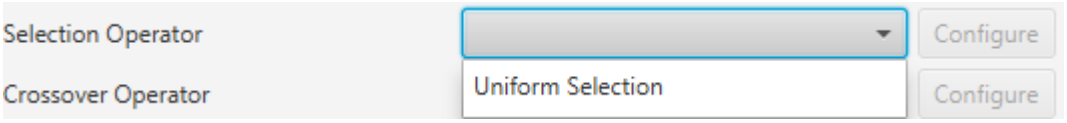

Ilustración 20, ComboBox expandido para configurar el operador.

El botón *Configure* permite configurar los parámetros que recibe el constructor del operador. Para el caso del operador *UniformSelection*, su interfaz de configuración se muestra en la Ilustración 15.

@OperatorOption

Esta anotación permite indicar las alternativas de operadores que puede recibir un parámetro para una categoría de operador indicada por la anotación *@OperatorInput.*

Dentro de esta anotación existen varios elementos. Estos elementos son:

- *displayName:* Nombre del operador. Este es el nombre visualizado en el *ComboBox* como se muestra en la ilustración 20.
- *value*: Arreglo que recibe instancias del tipo Class.

@FileInput

Esta anotación indica que hay un parámetro que espera recibir un objeto de tipo *File*. Cuando esta anotación está presente junto con su parámetro, en la interfaz, aparecerá un apartado que abre un *FileChooser* para buscar un archivo.

Dentro de esta anotación existen solo un elemento. Este es:

- *displayName:* Nombre del parámetro. Este nombre también corresponde al nombre visualizado en la ventana de configuración como muestra la ilustración 21.

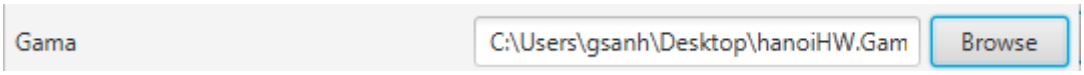

Ilustración 21, apartado para configurar el parámetro de tipo *File*

Si el TextField donde se muestra la ruta esta vacío, es decir, no se ha seleccionado un archivo, entonces será inyectado null en el parámetro correspondiente.

@NumberInput

Esta anotación indica que hay un parámetro del tipo *int* o *double* o de sus tipos envoltorios *Integer* o *Double*, respectivamente. Esta anotación agrega en la interfaz un *TextField* que solo permite como entrada un número. Si el tipo del parámetro es *int* o *Integer*, entonces el TextField solo permitirá ingresar números enteros. Por otro lado, si el parámetro es *double* o *Double*, entonces en la interfaz se podrá ingresar números enteros o decimales. El apartado para esta anotación se muestra en la Ilustración 22.

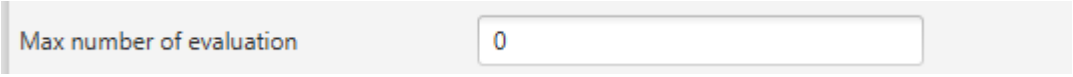

Ilustración 22, *TextField* presente cuando esta la anotación *@NumberInput*

Dentro de esta anotación existen solo un elemento. Este es:

- *displayName:* Nombre del parámetro.

@NumberToggleInput

Esta anotación indica que hay un conjunto de parámetros que son mutuamente excluyentes entre ellos, es decir, que solo un parámetro puede recibir el valor. En la interfaz, estarán bordeados los componentes que configuran la entrada de un mismo grupo. Dentro de un mismo grupo solo se podrá configurar un parámetro. El parámetro por configurar debe ser indicado activando el *ToggleButton* correspondiente, lo cual conllevara a la activación del *TextField*.

Dentro de esta anotación existen varios elementos. Estos elementos son:

- *groudID: String* con un id para el grupo. Las anotaciones *NumberToggerInput* que tengan el mismo id, en la interfaz, estarán en un apartado bordeado. Esto se aprecia en la Ilustración 23.
- *displaysNames*: Arreglo que recibe instancias del tipo Class.

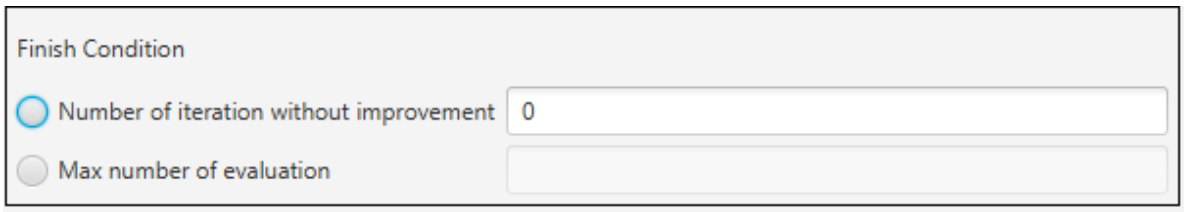

Ilustración 23, Apartado para *NumberToggleInput* con el mismo GroupID

El parámetro configurado en la interfaz recibirá el valor indicado en el *TextField*. Si este *TextField* queda vacío entonces recibirá el valor cero. En cambio, los demás parámetros recibirán el valor *Double.NEGATIVE_INFINITY*, si el parámetro era de tipo *double* o *Double*; o Integer.*NEGATIVE*_INFINITY si el parámetro era de tipo *int* o *Integer*. Por ejemplo, en la Ilustración 23, se observa que el parámetro *"Number of iteration without improvement"* esta activado, pero no contiene un valor, entonces al crear la instancia el constructor va a recibir el valor cero. Pero el parámetro "*Max number of evaluation",* al no haber sido escogido, recibirá el valor *Integer.NEGATIVE_INFITITY*, puesto que este parámetro era de tipo *int o* Integer.

En la Ilustración 24 se puede ver el mismo constructor presentado en la Ilustración 17, pero modificado para que sus dos últimos parámetros sean usando la anotación *NumberToggleInput.* Como se aprecia en la Ilustración 23, el *groupID* es usado para nombrar la sección.

```
@NewProblem(displayName = "Cost problem")
@Parameters(operators = {
        @OperatorInput(displayName = "Selection Operator", value = {
                @OperatorOption(displayName = "Uniform Selection", value = UniformSelection.class) }),
        @OperatorInput(displayName = "Crossover Operator", value = {
                @OperatorOption(displayName = "Integer SBX Crossover", value = IntegerSBXCrossover.class),
                @OperatorOption(displayName = "Integer Single Point Crossover", value = IntegerSinglePointCrossover.class) }), //
        @OperatorInput(displayName = "Mutation Operator", value = {
                @OperatorOption(displayName = "Integer Polynomial Mutation", value = IntegerPolynomialMutation.class),
                @OperatorOption(displayName = "Integer Range Random Mutation", value = IntegerRangeRandomMutation.class),
                @OperatorOption(displayName = "Integer Simple Random Mutation", value = IntegerSimpleRandomMutation.class) }) }, //
        files = { @FileInput(displayName = "Gama") }, //
        numbersToggle = { @NumberToggleInput(groupID = "Finish Condition", displayName = "Number of iteration without improvement"),
                @NumberToggleInput(groupID = "Finish Condition", displayName = "Max number of evaluation") })
public InversionCostRegister(Object selectionOperator, Object crossoverOperator, Object mutationOperator, File gama,
        int numberWithoutImprovement, int maxEvaluations)
```

Ilustración 24, constructor de clase que hereda de registrable y sus metadatos para cada parámetro, modificado para usar la anotación *NumberToggleInput*

En el elemento *numbersToggle* de la anotación *@Parameters*, las anotaciones que pertenezcan al mismo grupo deben estar continuas. En caso de que esto no se cumpla se lanzara una excepción al momento de ejecutar la aplicación.

Las anotaciones presentadas en las dos secciones anteriores deben ser usadas en constructores públicos.

5.2 Interfaz Registrable

Esta interfaz declara un único método llamado *build*, cuya declaración corresponde a la siguiente:

```
Algorithm<?> build(String inpPath) throws Exception;
```

Las clases de los objetos que implementen esta interfaz deben ser guardados en una estructura de datos, la cual será recorrida cuando se inicie la ejecución del programa y analizada usando la *Java Reflection API*. Este análisis consistirá en escanear y validar el cumplimiento de la convención establecida para las clases que implementan está interfaz. Esta convención consiste en lo siguiente:

- La clase debe contener un único constructor que use la anotación *@NewProblem.*
- Si el constructor requiere parámetros estos deben estas descritos usando la anotación *@Parameters*.
- El constructor debe declarar los parámetros en el siguiente orden, de acuerdo con su tipo.
    1. *Object:* Usado para inyectar los operadores. Estos después pueden ser casteados a su tipo correcto. La anotación correspondiente es @OperatorInput
    2. *File:* Usados cuando el problema requiere información adicional que se encuentra en un archivo diferente. La anotación correspondiente es @FileInput
    3. *int*, *Integer*, *double* o *Double:* Usado generalmente para configurar valores en el algoritmo o si el problema requiere otros valores que no fueron solicitados en al crear los operadores. Las anotaciones correspondientes son @NumberInput y @NumberToggleInput.
    4. El constructor debe solicitar la misma cantidad de parámetros que las descritas en la anotación @Parameters.

Si estas convenciones no se cumplen, entonces un error en tiempo de compilación será emitido como se menciono anteriormente en la sección anterior.

El orden en el que son inyectados los parámetros consiste en el siguiente:

1. Parámetros descritos por *@OperatorInput*
2. Parámetros descritos por *@FileInput*
3. Parámetros descritos por *@NumberInput*
4. Parámetros descritos por *@NumberToggleInput*

Una vez que se haya configurado el problema a través de la interfaz se creará la instancia de la clase que hereda de registrable y se llamara a su método *build*, para crear el algoritmo y comenzar su ejecución.

La estructura de datos en la que guardar la clase Registrable se encuentra en la clase ProblemRegistrar.

A continuación, se muestra un ejemplo de una clase que cumple todas las convenciones anteriores y hace uso de todos los parámetros:

```java
public class InversionCostRegister implements Registrable {
    private SelectionOperator<List<IntegerSolution>, List<IntegerSolution>> selection;
    private CrossoverOperator<IntegerSolution> crossover;
    private MutationOperator<IntegerSolution> mutation;
    private File gama;
    private int minPressure;
    private int populationSize;
    private int numberWithoutImprovement;
    private int maxEvaluations;

    @NewProblem(displayName = "Cost problem")
    @Parameters(operators = {
            @OperatorInput(displayName = "Selection Operator", value = {
                    @OperatorOption(displayName = "Uniform Selection", value = UniformSelection.class) }),
            @OperatorInput(displayName = "Crossover Operator", value = {
                    @OperatorOption(displayName = "Integer SBX Crossover", value = IntegerSBXCrossover.class),
                    @OperatorOption(displayName = "Integer Single Point Crossover", value = IntegerSinglePointCrossover.class) }), //
            @OperatorInput(displayName = "Mutation Operator", value = {
                    @OperatorOption(displayName = "Integer Polynomial Mutation", value = IntegerPolynomialMutation.class),
                    @OperatorOption(displayName = "Integer Range Random Mutation", value = IntegerRangeRandomMutation.class),
                    @OperatorOption(displayName = "Integer Simple Random Mutation", value = IntegerSimpleRandomMutation.class) }) }, //
            files = { @FileInput(displayName = "Gama") }, //
            numbers = { @NumberInput(displayName = "Min Pressure"),
                    @NumberInput(displayName = "Population Size") },
            numbersToggle = { @NumberToggleInput(groupID = "Finish Condition", displayName = "Number of iteration without improvement"),
                    @NumberToggleInput(groupID = "Finish Condition", displayName = "Max number of evaluation") })
    @SuppressWarnings("unchecked")//The object injected are indicated in operators elements. It guarantee its types.
    public InversionCostRegister(Object selectionOperator, Object crossoverOperator, Object mutationOperator,
            File gama, int minPressure, int populationSize
            int numberWithoutImprovement, int maxEvaluations) {
        this.selection = (SelectionOperator<List<IntegerSolution>, List<IntegerSolution>>) selectionOperator;
        this.crossover = (CrossoverOperator<IntegerSolution>) crossoverOperator;
        this.mutation = (MutationOperator<IntegerSolution>) mutationOperator;
        this.gama = gama;
        this.minPressure = minPressure;
        this.populationSize = populationSize;
        this.numberWithoutImprovement = numberWithoutImprovement;
        this.maxEvaluations = maxEvaluations;
    }

    @Override
    public Algorithm<IntegerSolution> build(String inpPath) throws Exception {
        if (inpPath == null || inpPath.isEmpty()) {
            throw new ApplicationException("There isn't a network opened");
        }
        EpanetAPI epanet;
        GeneticAlgorithm2<IntegerSolution> algorithm = null;
        epanet = new EpanetAPI();
        epanet.ENopen(inpPath, "ejecucion.rpt", "");

        if (this.gama == null) {
            throw new ApplicationException("There isn't gama file");
        }
        Problem<IntegerSolution> problem = new InversionCostProblem(epanet, this.gama.getAbsolutePath(), minPressure);
        algorithm = new GeneticAlgorithm2<IntegerSolution>(problem, populationSize, selection, crossover, mutation);

        if (Integer.compare(numberWithoutImprovement, Integer.NEGATIVE_INFINITY) != 0){
            algorithm.setMaxNumberOfIterationWithoutImprovement(numberWithoutImprovement);
        }
        else if (Integer.compare(maxEvaluations, Integer.NegativeInfinity) != 0){
            algorithm.setMaxEvaluations(maxEvaluations);
        }
        return algorithm;
    }
}
```

Ilustración 25, Ejemplo de cómo debería lucir la clase que hereda de Registrable

4 Matriz de trazado

A continuación se muestra la matriz de trazado de Requisitos de Software vs. Módulos. Cabe señalar que producto de las modificaciones que los módulos han sufrido desde el comienzo del proyecto, la numeración de los mismos en ReqAdmin difiere de la empleada en éste documento. Es por esto que se adjunta una tabla mostrando las equivalencias entre ambas notaciones.

Tabla 1, matriz de trazado RS vs. MD

	MD0015	MD0016	MD0017	MD0018	MD0019	MD0020	MD0021	MD0022	MD0023	MD0024	MD0025	MD0027	MD0028	MD0029	MD0030	MD0031	MD0032
RS0002			X	X													
RS0003			X	X													
RS0004			X	X													
RS0005						X	X										
RS0006									X								
RS0007						X	X	X									X
RS0008	X	X															
RS0009									X								
RS0010								X						X			
RS0011								X		X							
RS0012												X					
RS0013												X					
RS0014					X												
RS0015															X		
RS0016															X		
RS0017																X	
RS0018								X									
RS0019													X				
RS0020													X				
RS0021						X	X	X									
RS0022											X						
RS0023								X									
RS0024								X									
RS0026								X						X			
RS0027						X	X										
RS0029								X									