



**UNIVERSIDAD DE TALCA**  
**FACULTAD DE INGENIERÍA**  
**ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN**

**Manual de usuario**

**JHAWANET FRAMEWORK**

## TABLA DE CONTENIDOS

página

### Tabla de Contenidos

Índice de Figuras	1
1. Como agregar un nuevo algoritmo	2
2. Como agregar un nuevo problema	4
3. Como agregar un nuevo operador	6
3.1. <i>@DefaultConstructor</i> . . . . .	6
4. Como agregar un nuevo experimento	7
4.1. Interfaz Registrable . . . . .	9
4.2. <i>@NewProblem</i> . . . . .	13
4.3. <i>@Parameters</i> . . . . .	15
4.4. <i>@OperatorInput</i> . . . . .	16
4.5. <i>@OperatorOption</i> . . . . .	17
4.6. <i>@FileInput</i> . . . . .	17
4.7. <i>@NumberInput</i> . . . . .	18
4.8. <i>@NumberToggleInput</i> . . . . .	19
Glosario	21

## ÍNDICE DE FIGURAS

	página
1. Diagrama de clases del módulo de metaheurísticas. . . . .	2
2. Mensaje retornado por el método <i>getStatusOfExecution</i> en la ventana monoobjetivo. . . . .	3
3. Mensaje retornado por el método <i>getStatusOfExecution</i> en la ventana multiobjetivo. . . . .	3
4. Método <i>runASingleStep</i> para los algoritmos evolutivos. . . . .	4
5. Ejemplo del método <i>applySolutionToNetwork</i> . . . . .	5
6. Constructor de un solo parámetro. . . . .	6
7. Constructor con varios parámetros. . . . .	6
8. Interfaz para configurar el operador <i>UniformSelection</i> . . . . .	7
9. Interfaz para configurar el operador <i>IntegerPolynomialMutation</i> . . . . .	7
10. Diagrama módulo metaheurística extendido. . . . .	8
11. Jerarquía de clases de la interfaz Registrable. . . . .	9
12. Constructor de la clase <i>PipeOptimizingRegister</i> . . . . .	11
13. Implementación del método build de la clase <i>PipeOptimizingRegister</i> . . . . .	12
14. Implementación del método opcional getParameters de la clase <i>Pi- peOptimizingRegister</i> . . . . .	13
15. Constructor de clase que hereda de registrable y sus metadatos para cada parámetro. . . . .	14
16. Menú de problemas. . . . .	15
17. Componente <i>ComboBox</i> para configurar los operadores. . . . .	16
18. <i>ComboBox</i> expandido para configurar el operador. . . . .	17
19. Apartado para configurar el parámetro de tipo <i>File</i> . . . . .	18
20. <i>TextField</i> presente cuando está la anotación <i>@NumberInput</i> . . . . .	18
21. Apartado para <i>NumberToggleInput</i> con el mismo GroupID. . . . .	19

## 1. Como agregar un nuevo algoritmo

Para agregar un nuevo algoritmo a la aplicación se debe implementar la interfaz *Algorithm*. En la Figura 1 se puede ver la interfaz correspondiente.

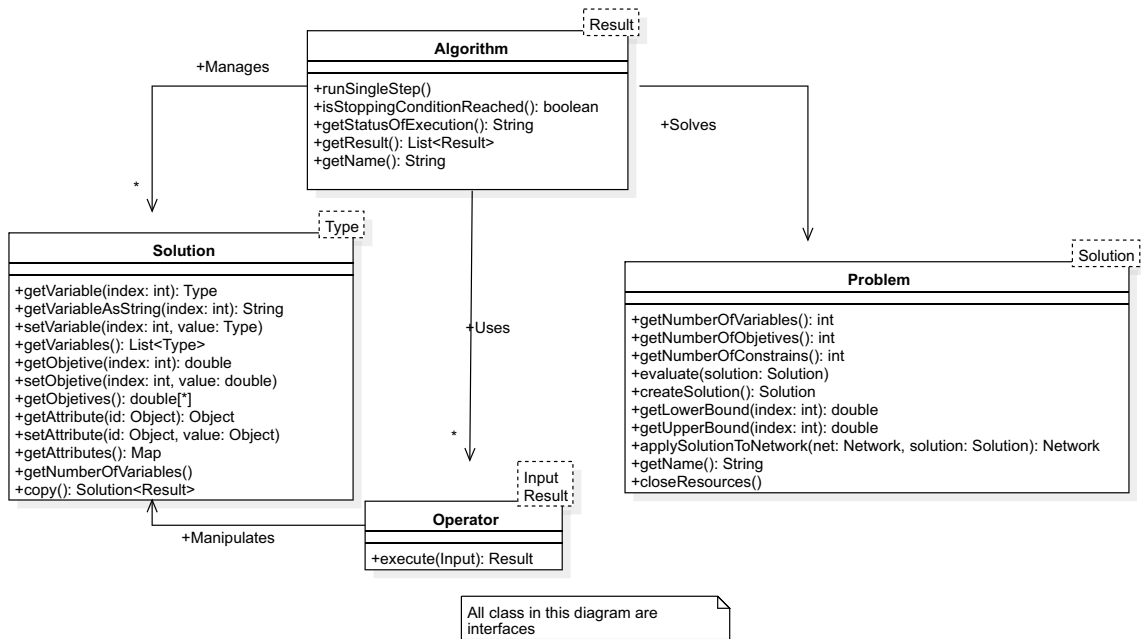


Figura 1: Diagrama de clases del módulo de metaheurísticas.

Esta interfaz cuenta con los siguientes métodos:

- *RunSingleStep*: Este método debe ejecutar un único paso del algoritmo (Una sola generación/iteración).
- *isStoppingConditionReached*: Este método indica si la condición de término del algoritmo ha sido alcanzada.
- *getStatusOfExecution*: Este método puede devolver un *String* con información del algoritmo como se muestra en la Figura 2 y 3.
- *getResult*: Devuelve el resultado del algoritmo.
- *getName*: Devuelve el nombre del algoritmo.

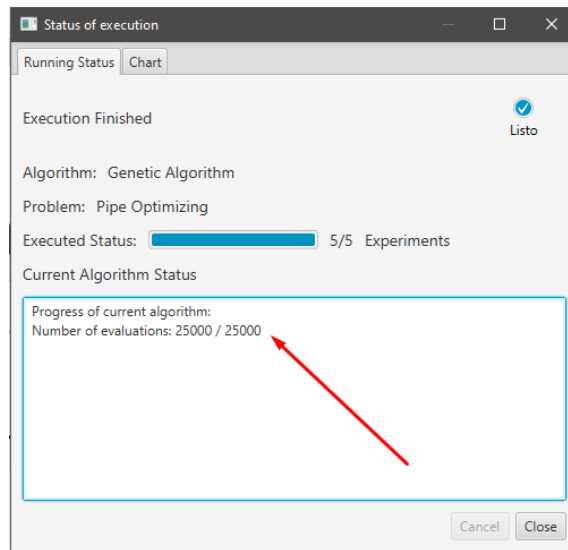


Figura 2: Mensaje retornado por el método *getStatusOfExecution* en la ventana monoobjetivo (Está indicado por la flecha roja).

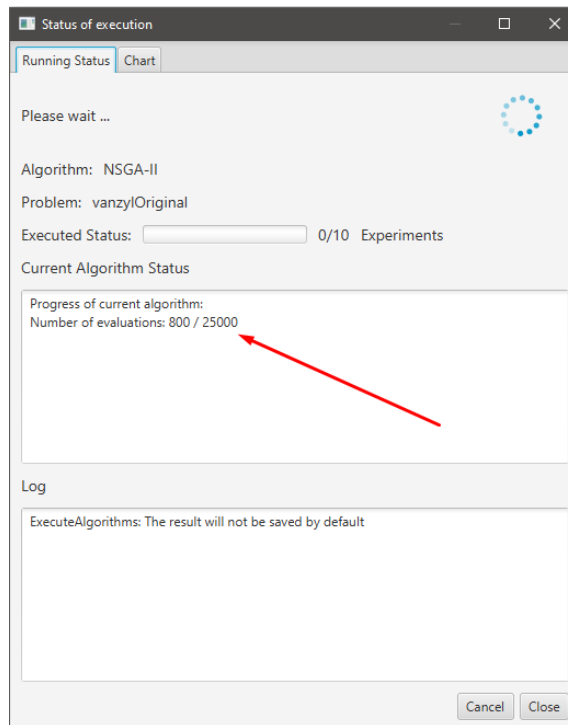


Figura 3: Mensaje retornado por el método *getStatusOfExecution* en la ventana multiobjetivo (Está indicado por la flecha roja).

Para llevar a cabo la simulación, la aplicación llama al método *RunSingleStep* hasta que *isStoppingConditionReached* sea verdadero. Adicionalmente, dentro del mismo método *RunSingleStep* también se debería asegurar que una vez sea alcanzada la condición de termino no se realicen nuevas operaciones. Un ejemplo de *RunSingleStep* utilizado en el Algoritmo NSGAI y GA se puede ver en la Figura 4.

```
int step = 0;
List<IntegerSolution> population;
public void runSingleStep() throws Exception, EpanetException {
    List<S> offspringPopulation;
    List<S> selectionPopulation;
    // durante la primera iteración inicializa la población
    if (step == 0) {
        population = createInitialPopulation();
        population = evaluatePopulation(population);
        initProgress();
    }

    if (!isStoppingConditionReached()) {
        selectionPopulation = selection(population);
        offspringPopulation = reproduction(selectionPopulation);
        offspringPopulation = evaluatePopulation(offspringPopulation);
        population = replacement(population, offspringPopulation);
        updateProgress();
    }

    this.step++;
}
```

Figura 4: Método *runASingleStep* para los algoritmos evolutivos.

## 2. Como agregar un nuevo problema

Para agregar un nuevo problema se debe implementar la interfaz *Problem* que se muestra en la Figura 1. Esta interfaz posee los siguientes métodos:

- *getNumberOfVariables*: Indica el número de variables del problema.
- *getNumberOfObjectives*: Indica el número de objetivos del problema.
- *getNumberOfConstrains*: Indica el número de restricciones del problema.
- *evaluate*: Evalúa una solución y sus restricciones.

- *createSolution*: Crea una nueva solución para el problema.
- *getLowerBound*: Indica el valor mínimo que puede tomar una variable en un índice específico.
- *getUpperBound*: Indica el valor máximo que puede tomar una variable en un índice específico.
- *applySolutionToNetwork*: Toma una solución y la aplica sobre una red (Instancia de *Network* que posteriormente puede ser guardada como un archivo inp). Este método es opcional y en caso de que no esté implementado devuelve el valor *null*. Se puede ver un ejemplo de este método en la Figura 5. Este método es llamado por la aplicación cuando se selecciona una solución en la pestaña de resultados y se presiona el botón guardar como inp.
- *getName*: El nombre del problema.
- *closeResources*: Cierra los recursos usado por el problema. Generalmente, el único recurso a cerrar sería el simulador hidráulico EpanetAPI. Implementar este método es opcional. Este método es llamado por la aplicación cuando se termina el experimento.

```

public Network applySolutionToNetwork(Network network, Solution<?> solution) {
    // El objeto solución posee índices que van desde 1 hasta la cantidad de diámetros posibles.
    IntegerSolution iSolution = (IntegerSolution) solution;
    Collection<Link> links = network.getLinks();
    int i = 0;
    for (Link link : links) {
        /*
         * Obtiene el diámetro correspondiente a un índice específico.
         */
        double diameter = this.gamas.get(iSolution.getVariable(i) - 1).getDiameter();
        /*
         * Reemplaza en la tubería y válvula el diámetro. Si en enlace es una bomba se ignora.
         */
        if (link instanceof Pipe) {
            Pipe pipe = (Pipe) link;
            pipe.setDiameter(diameter);
        } else if (link instanceof Valve) {
            Valve valve = (Valve) link;
            valve.setDiameter(diameter);
        }
        i++;
    }
    return network;
}

```

Figura 5: Ejemplo del método *applySolutionToNetwork*.

### 3. Como agregar un nuevo operador

Para crear un nuevo operador se debe implementar la interfaz *Operator* o alguna de sus subinterfaces o subclases. Esta interfaz cuenta con un único método. La interfaz se muestra en la Figura 1. El método de la interfaz es:

- *execute*: Método que realiza una operación sobre un operando y devuelve un objeto resultante de dicha operación. Generalmente, este método realiza una acción sobre una solución o una lista de soluciones y retorna la solución resultante de la operación realizada.

Para configurar los parámetros del operador desde la ventana de configuración, debe haber un constructor público que posea la anotación *@DefaultConstructor*.

#### 3.1. *@DefaultConstructor*

La anotación *@DefaultConstructor* indica el constructor que debe ser usado al momento de crear una instancia del operador. Esta anotación recibe un arreglo de *@NumberInput*, el cual se define en la siguiente sección. El arreglo debe tener la misma cantidad de argumentos que los parámetros del constructor como se muestra en la Figura 6 y Figura 7.

```
@DefaultConstructor(@NumberInput(displayName = "constant", defaultValue = 1.5))
public UniformSelection(double constant)
```

Figura 6: Constructor de un solo parámetro.

```
@DefaultConstructor({
    @NumberInput(displayName = "MutationProbability", defaultValue = 0.1),
    @NumberInput(displayName = "DistributionIndex", defaultValue = 0.1
})
public IntegerPolynomialMutation(double mutationProbability, double distributionIndex)
```

Figura 7: Constructor con varios parámetros.

Esta anotación solo puede ser usada en un único constructor por clase. Usar esta anotación en más 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 gráfica creada a partir de las anotaciones de la Figura 6 y 7 se puede ver en la Figura 8 y 9, respectivamente.



Figura 8: Interfaz para configurar el operador *UniformSelection*.

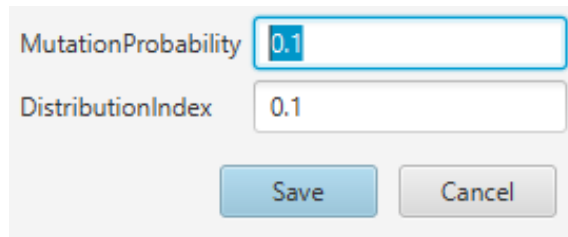


Figura 9: Interfaz para configurar el operador *IntegerPolynomialMutation*.

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

## 4. Como agregar un nuevo experimento

Un experimento es una clase que contiene una lista de algoritmos para resolver un problema específico. Los algoritmos que conforman el experimento deben ser de un mismo tipo, por ejemplo, “n” repeticiones del Algoritmo Genético. Actuando “n” como el número de ejecuciones independientes del algoritmo. En la Figura 10, se muestra las interfaces del módulo de metaheurísticas extendido.

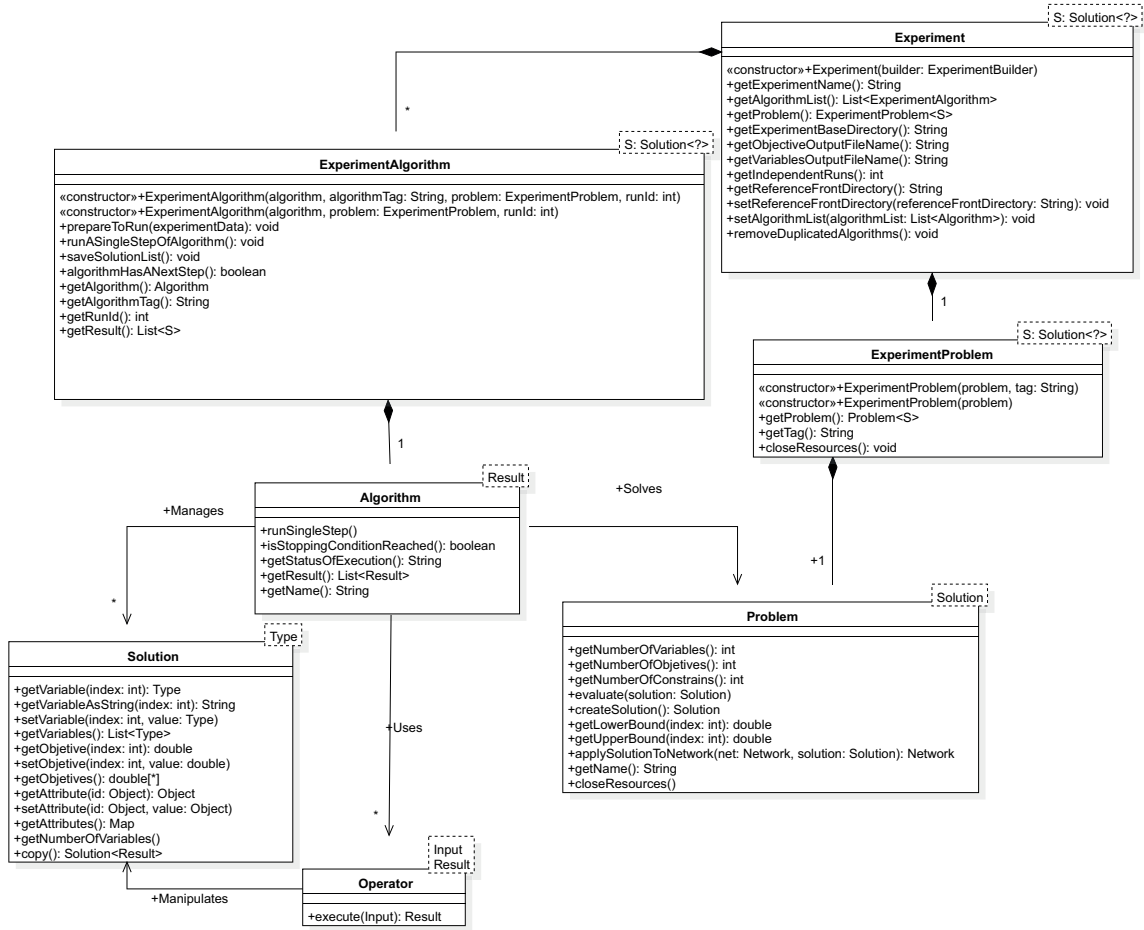


Figura 10: Diagrama módulo metaheurística extendido.

Para crear el experimento la aplicación cuenta con una clase llamada *ExperimentBuilder*. Los parámetros *ExperimentBaseDirectory* y *ReferenceFrontDirectory* solo tienen uso en experimentos multiobjetivos. *ExperimentBaseDirectory* indica donde guardar la frontera de Pareto de cada repetición del algoritmo multiobjetivo. En cambio, *ReferenceFrontDirectory* indica donde guardar la frontera de Pareto final, que se obtiene al unir las fronteras de cada repetición y extraer solamente las soluciones no dominadas de dicho conjunto.

#### 4.1. Interfaz Registrable

Con el fin de agregar un nuevo experimento en la aplicación, se debe crear una clase que herede de *SingleObjectiveRegistrable* o *MultiObjectiveRegistrable*, dependiendo de si el problema es monoobjetivo o multiobjetivo, respectivamente. Las clases que heredan de *Registrable* funcionan como *templates* para crear nuevos experimentos. En la Figura 11, se presenta la jerarquía de clases de la interfaz *Registrable*.

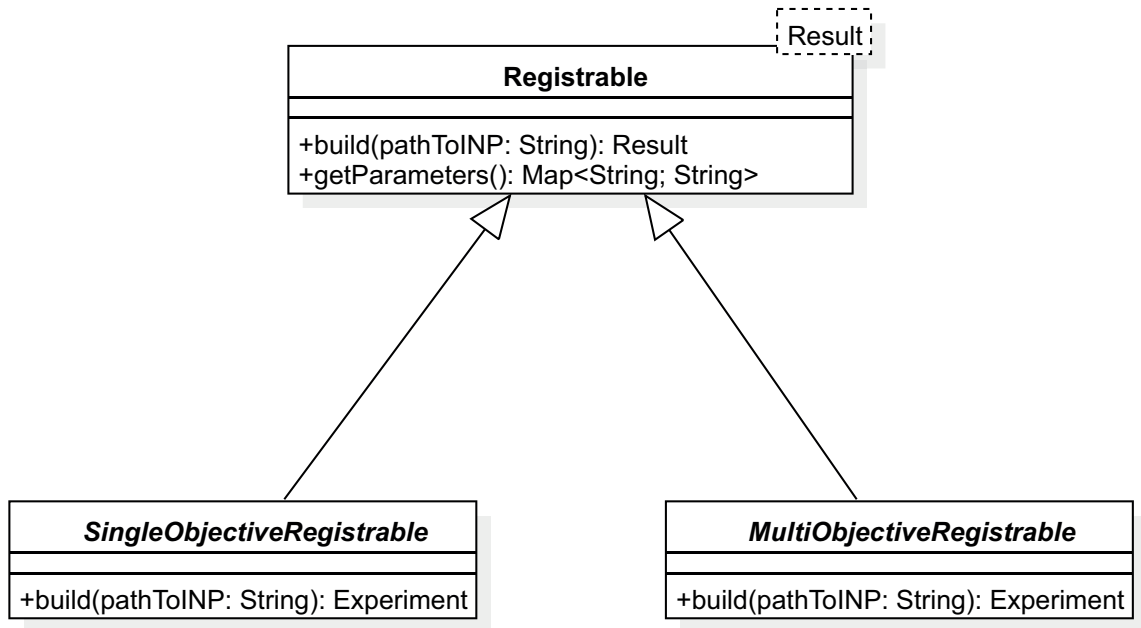


Figura 11: Jerarquía de clases de la interfaz Registrable.

Ambas interfaces devuelven como resultado una instancia del experimento la cual es usada para realizar la simulación. Los métodos utilizados por la interfaz *Registrable* son:

- *build*: Construye un experimento.
- *getParameters*: Este método devuelve un par clave valor, el cual es usado para agregar columnas extras a la ventana de resultados. La clave es usada como el nombre de la columna, mientras que el valor es usado como el valor de la columna. Este método es opcional.

En el constructor público de la clase que herede de una de las subinterfaces de *Registrable*, se debe utilizar la anotación *@NewProblem*. Adicionalmente, si se quiere mostrar la interfaz de configuración, dicho constructor debe incluir la anotación *@Parameters*.

Las clases que implementen cualquiera de las dos interfaces mencionadas anteriormente deben ser guardados en una estructura de datos, la cual es 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 esta interfaz. Esta convención consiste en lo siguiente:

- La clase debe contener un único constructor que use la anotación *@NewProblem*.
- Si el constructor requiere parámetros éstos deben estar 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 parámetros pueden posteriormente 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 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.

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 llamará a su método *build*, para crear el experimento y comenzar su ejecución.

La estructura de datos para registrar las clases que heredan de *SingleObjectiveRegistrable* y *MonoObjectiveRegistrable* se encontrará en la clase *RegistrableConfiguration*.

En las Figura 12, 13, 14, se muestran un ejemplo del constructor, uno del método *build* y uno del método *getParameters*, utilizado en el problema de optimización de los costos de las tuberías.

```
@NewProblem(displayName = "Pipe optimizing", algorithmName = "Genetic Algorithm",
description = "The objective of this problem is to optimize the cost of "
"construction of the network by varying the diameter of the pipe in order "
"to ensure a minimum level of pressure.")
@Parameters(operators = {
    @OperatorInput(displayName = "Selection Operator", value = {
        @OperatorOption(displayName = "Uniform Selection", value = UniformSelection.class)
    }),
    @OperatorInput(displayName = "Crossover Operator", value = {
        @OperatorOption(displayName = "Integer Single Point Crossover", value = IntegerSinglePointCrossover.class),
        @OperatorOption(displayName = "Integer SEX Crossover", value = IntegerSEXCrossover.class)
    }), //
    @OperatorInput(displayName = "Mutation Operator", value = {
        @OperatorOption(displayName = "Integer Simple Random Mutation", value = IntegerSimpleRandomMutation.class),
        @OperatorOption(displayName = "Integer Polynomial Mutation", value = IntegerPolynomialMutation.class),
        @OperatorOption(displayName = "Integer Range Random Mutation", value = IntegerRangeRandomMutation.class)
    }), //
    files = {
        @FileInput(displayName = "Gama")
    }, //
    numbers = {
        @NumberInput(displayName = "Independent run", defaultValue = 5),
        @NumberInput(displayName = "Min pressure", defaultValue = 30),
        @NumberInput(displayName = "Population Size", defaultValue = 1000)
    }, //
    numbersToggle = {
        @NumberToggleInput(groupID = "Finish Condition", displayName = "Number of iteration without improvement", defaultValue = 100),
        @NumberToggleInput(groupID = "Finish Condition", displayName = "Max number of evaluation", defaultValue = 1000)
    })
public PipeOptimizingRegister(Object selectionOperator, Object crossoverOperator, Object mutationOperator, File gama, int independentRun,
int minPressure, int populationSize, int numberWithoutImprovement, int maxEvaluations) throws Exception
```

Figura 12: Constructor de la clase *PipeOptimizingRegister*.

```

@Override
public Experiment<IntegerSolution> build(String inpPath) throws Exception {
    if (inpPath == null || inpPath.isEmpty()) {
        throw new ApplicationException("There isn't a network opened");
    }
    // Crea el simulador hidráulico.
    EpanetAPI epanet = new EpanetAPI();
    /*
     * Inicializa el simulador. Es importante no llamar a epanet.ENclose() hasta que termina la ejecución
     * del experimento. Es por ello, que la llamada a epanet.ENclose() debe ir en el método closeResource del problema,
     * puesto que este método se llama cuando termina la ejecución del algoritmo.
     */
    epanet.ENopen(inpPath, "ejecucion.rpt", "");

    // El archivo gama, es un archivo que posee los diámetros disponibles de la red y el costo de éstos.
    if (this.gama == null) {
        throw new ApplicationException("There isn't gama file");
    }
    this.problem = new PipeOptimizing(epanet, this.gama.getAbsolutePath(), this.minPressure);

    // Indica el problema a usar por el experimento
    ExperimentProblem<IntegerSolution> experimentProblem = new ExperimentProblem<>(this.problem);

    /*
     * Crea varios repeticiones del mismo algoritmo. En este caso, son del algoritmos genético.
     * El número de algoritmos genéticos a crear corresponde al valor de la variable independentRun.
     */
    List<ExperimentAlgorithm<IntegerSolution>> experimentAlgorithms = ExperimentUtils.configureAlgorithmList(experimentProblem,
        this.independentRun,
        () -> {
            GeneticAlgorithmBuilder builder = new GeneticAlgorithmBuilder(this.problem, this.crossover, this.mutation)
                .setCrossoverOperator(crossover);
            if (this.numberWithoutImprovement != Integer.MIN_VALUE) {
                builder.setMaxNumberOfEvaluationWithoutImprovement(this.numberWithoutImprovement);
            } else {
                builder.setMaxEvaluations(this.maxEvaluations);
            }
            return builder.build();
        });

    // Configura el experimento
    Experiment<IntegerSolution> experiment = new ExperimentBuilder<IntegerSolution>("PipeOptimizing")
        .setIndependentRuns(this.independentRun)
        .setAlgorithmList(experimentAlgorithms)
        .setProblem(experimentProblem)
        .build();

    return experiment;
}

```

Figura 13: Implementación del método build de la clase *PipeOptimizingRegister*.

```

@Override
public Map<String, String> getParameters() {
    Map<String, String> map = new LinkedHashMap<>();
    map.put("Min Pressure", "" + this.minPressure);
    map.put("Population Size", "" + this.populationSize);
    // see if number without improvement was configure or not
    if (this.numberWithoutImprovement != Integer.MIN_VALUE) {
        map.put("Number without improvement", "" + this.numberWithoutImprovement);
    } else {
        map.put("Number of max evaluations", "" + this.maxEvaluations);
    }

    // for selection
    if (this.selection instanceof UniformSelection) {
        map.put("Selection", "UniformSelection");
        map.put("Uniform Selection Constant", "" + ((UniformSelection<IntegerSolution>) this.selection).getConstant());
    }

    // for crossover
    if (this.crossover instanceof IntegerSBXCrossover) {
        map.put("Crossover", "IntegerSBXCrossover");
        map.put("Crossover Probability", "" + ((IntegerSBXCrossover) this.crossover).getCrossoverProbability());
        map.put("Crossover Distribution Index", "" + ((IntegerSBXCrossover) this.crossover).getDistributionIndex());
    } else if (this.crossover instanceof IntegerSinglePointCrossover) {
        map.put("Crossover", "IntegerSinglePointCrossover");
        map.put("Crossover Probability", "" + ((IntegerSinglePointCrossover) this.crossover).getCrossoverProbability());
    }

    // for mutation
    if (this.mutation instanceof IntegerPolynomialMutation) {
        map.put("Mutation", "IntegerPolynomialMutation");
        map.put("Mutation Probability", "" + ((IntegerPolynomialMutation) this.mutation).getMutationProbability());
        map.put("Mutation Distribution Index", "" + ((IntegerPolynomialMutation) this.mutation).getDistributionIndex());
    } else if (this.mutation instanceof IntegerSimpleRandomMutation) {
        map.put("Mutation", "IntegerSimpleRandomMutation");
        map.put("Mutation Probability", "" + ((IntegerSimpleRandomMutation) this.mutation).getMutationProbability());
    } else if (this.mutation instanceof IntegerRangeRandomMutation) {
        map.put("Mutation", "IntegerRangeRandomMutation");
        map.put("Mutation Probability", "" + ((IntegerRangeRandomMutation) this.mutation).getMutationProbability());
        map.put("Mutation Range", "" + ((IntegerRangeRandomMutation) this.mutation).getRange());
    }

    return map;
}

```

Figura 14: Implementación del método opcional `getParameters` de la clase *PipeOptimizingRegister*.

A continuación se define detalladamente cada una de las anotaciones permitidas en la clase *Registrable*.

#### 4.2. *@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 Figura 15.

```

@NewProblem(displayName = "Pipe optimizing", algorithmName = "Genetic Algorithm",
description = "The objective of this problem is to optimize the cost of "
"construction of the network by varying the diameter of the pipe in order "
"to ensure a minimum level of pressure.")
@Parameters(operators = {
    @OperatorInput(displayName = "Selection Operator", value = {
        @OperatorOption(displayName = "Uniform Selection", value = UniformSelection.class)
    }),
    @OperatorInput(displayName = "Crossover Operator", value = {
        @OperatorOption(displayName = "Integer Single Point Crossover", value = IntegerSinglePointCrossover.class),
        @OperatorOption(displayName = "Integer SBX Crossover", value = IntegerSBXCrossover.class)
    }), //
    @OperatorInput(displayName = "Mutation Operator", value = {
        @OperatorOption(displayName = "Integer Simple Random Mutation", value = IntegerSimpleRandomMutation.class),
        @OperatorOption(displayName = "Integer Polynomial Mutation", value = IntegerPolynomialMutation.class),
        @OperatorOption(displayName = "Integer Range Random Mutation", value = IntegerRangeRandomMutation.class)
    }), //
    files = {
        @FileInput(displayName = "Gama")
    }, //
    numbers = {
        @NumberInput(displayName = "Independent run", defaultValue = 8),
        @NumberInput(displayName = "Min pressure", defaultValue = 30),
        @NumberInput(displayName = "Population Size", defaultValue = 1000)
    }, //
    numbersToggle = {
        @NumberToggleInput(groupId = "Finish Condition", displayName = "Number of iteration without improvement", defaultValue = 100),
        @NumberToggleInput(groupId = "Finish Condition", displayName = "Max number of evaluation", defaultValue = 1000)
    })
public PipeOptimizingRegister(Object selectionOperator, Object crossoverOperator, File gama, int independentRun,
    int minPressure, int populationSize, int numberWithoutImprovement, int maxEvaluations) throws Exception

```

Figura 15: Constructor de clase que hereda de registrable y sus metadatos para cada parámetro.



Los elementos en esta anotación consisten en:

- *displayName*: El nombre del problema. Este nombre también actúa como el nombre de la categoría.
- *algorithm*: Un *String* con el nombre del algoritmo usado para resolver el problema.
- *description*: Un *String* con la descripción del algoritmo.

El nombre dado en el elemento *displayName*, es el nombre visible del problema en el menú de la aplicación y permite agrupar a los problemas que tengan el mismo nombre, pero distintos algoritmos, como se ve en la Figura 16.

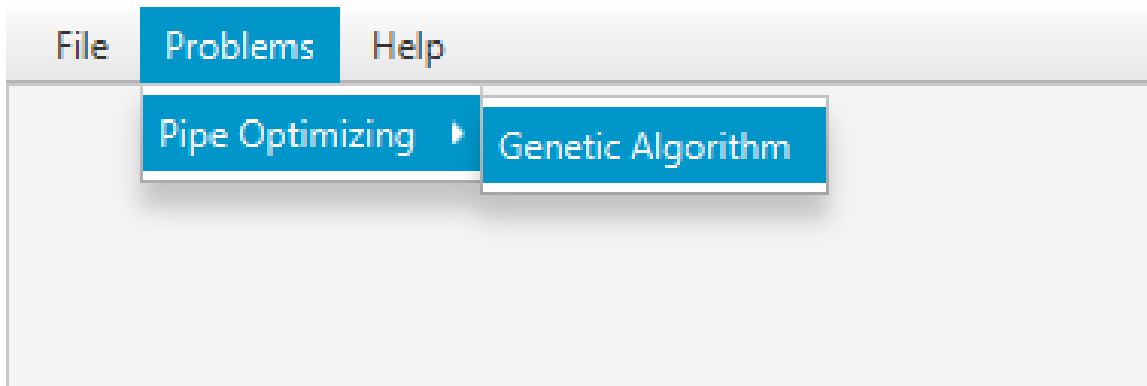


Figura 16: Menú de problemas.

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

#### 4.3. *@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 Figura 15.

#### 4.4. ***@OperatorInput***

Esta anotación agrega información a uno de los parámetros del constructor acerca de los posibles operadores que pueden ser recibidos por ese parámetro. Dentro de esta anotación existen varios elementos. Estos elementos son:

1. *displayName*: Nombre de categoría para los operadores.
2. *value*: Arreglo que recibe anotaciones del tipo *OperatorOption*.

En la ventana de configuración de problema, estos parámetros son vistos con un *ComboBox* como muestra la Figura 17.

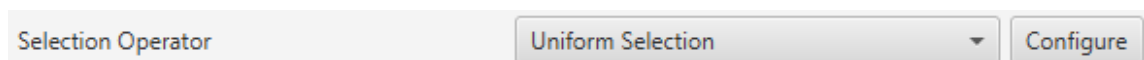


Figura 17: Componente *ComboBox* para configurar los operadores.

Las alternativas disponibles dentro del *ComboBox* están dadas por aquellas indicadas en el elemento *value* de este operador. Como se muestra en la Figura 15, la única alternativa para el *Selection Operator* es el operador *UniformSelection* apreciado en la Figura 18.

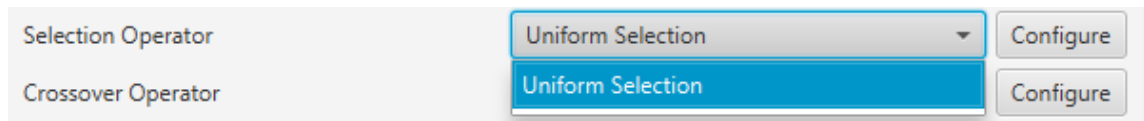


Figura 18: *ComboBox* expandido para configurar el operador.

Por defecto, el *ComboBox* selecciona el primer elemento de la lista. El botón *Configure* permite configurar los parámetros que recibe el constructor del operador, aquel que posee la anotación *@DefaultConstructor*. Para el caso del operador *UniformSelection*, su interfaz de configuración se muestra en la Figura 8.

#### 4.5. *@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 Figura 18.
- *value*: Instancia del tipo *Class* que referencia el tipo de operador.

#### 4.6. *@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* o un *DirectoryChooser* para buscar un archivo o directorio, respectivamente. Dentro de esta anotación existen varios elementos. Éstos son:

- *displayName*: Nombre del parámetro. Este nombre también corresponde al nombre visualizado en la ventana de configuración como se muestra en la Figura 19.

- *type*: Indica el modo en que se abrirá el *FileChooser*. Este elemento recibe un enumerado del tipo *Type*; los cuales son *Type.OPEN*, *Type.SAVE*, que abren un *FileChooser* para leer o guardar un archivo; y *Type.Directory*, el cual abre un *DirectoryChooser* para seleccionar un directorio. La opción por defecto es *FileType.OPEN*.

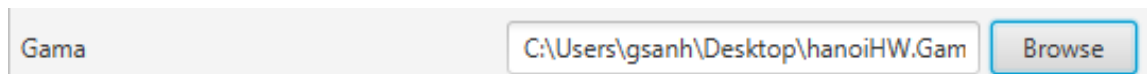


Figura 19: Apartado para configurar el parámetro de tipo *File*.

Si el *TextField* donde se muestra la ruta está vacío, es decir, no se ha seleccionado un archivo o carpeta, entonces será inyectado *null* en el parámetro correspondiente del constructor.

#### 4.7. *@NumberInput*

Esta anotación indica que hay un parámetro del tipo *int* o *double* o 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 permite ingresar números enteros. Por otro lado, si el parámetro es *double* o *Double*, entonces en la interfaz se acepta el ingreso de números enteros o decimales. El apartado para esta anotación se muestra en la Figura 20.

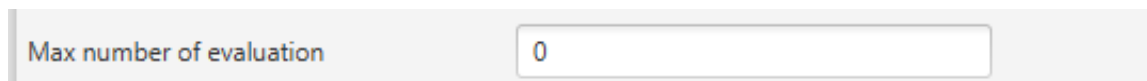


Figura 20: *TextField* presente cuando está la anotación *@NumberInput*.

Dentro de esta anotación existen los siguientes elementos:

- *displayName*: Nombre del parámetro.
- *defaultValue*: Valor por defecto de la propiedad. Si el tipo de parámetro en el constructor de la clase que hereda de *Registrable* es un entero, pero se ingresa

como valor por defecto un número con decimales, los decimales serán truncados. Si este elemento no se define su valor por defecto es 0.

#### 4.8. *@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, el nombre del grupo aparece sobre los componentes. Dentro de un mismo grupo solo se puede configurar un parámetro. El parámetro por configurar debe ser indicado activando el *ToggleButton* correspondiente, lo cual conlleva 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, se encuentran en una sección cuyo título es el nombre del grupo. Esto se aprecia en la Figura 21.
- *displayName*: Nombre del parámetro.
- *defaultValue*: Valor por defecto de la propiedad. Si el tipo de parámetro en el constructor de la clase que hereda de *Registrable* es un entero, pero se ingresa como valor por defecto un número con decimales, los decimales serán truncados. Si este elemento no se define su valor por defecto es 0.

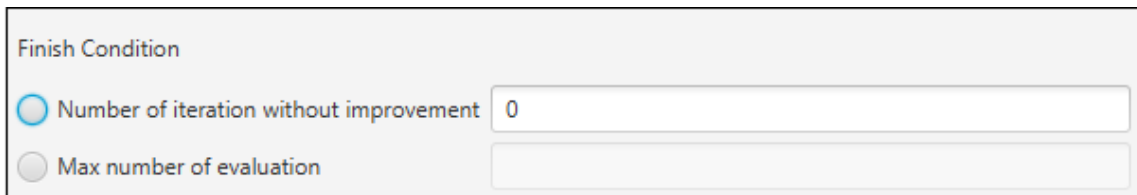


Figura 21: Apartado para *NumberToggleInput* con el mismo GroupID.

El parámetro configurado en la interfaz de usuario recibe el valor indicado en el *TextField*. Si el *TextField* queda vacío entonces recibe el valor cero. Sin embargo, los demás parámetros, cuyos *TextField* están deshabilitados, van a recibir el

valor ***Double.MIN\_VALUE***, si el parámetro es de tipo *double* o *Double*; o ***Integer.MIN\_VALUE*** si el parámetro es de tipo *int* o *Integer*. Por ejemplo, en la Figura 21, 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, recibe el valor *Integer.MIN\_VALUE*, puesto que este parámetro era de tipo *int* o *Integer*.

En la Figura 15 se puede ver el uso de la anotación *NumberToggleInput*. El componente que representa esta anotación en la GUI se puede apreciar en la Figura 21. El *groupID* es usado para nombrar la sección.

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 lanza una excepción al momento de ejecutar la aplicación.

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

**Glosario**

**GA** Algoritmo Genético

**NSGAI** Non-dominated Sorting Genetic Algorithm