Project: Solving proximity constraints

Jan-Michael Holzinger[*] Sophie Hofmanninger[†]

JKU Linz — SS2019

Version Number	Changes Summary	Author
0.1		Jan-Michael
0.2	added System Model	Jan-Michael
0.3	modified Parser, added Workflow	Jan-Michael
0.4	add ConstraintSimplification	Sophie
0.5	Proximity Relation (as Matrix) added	Jan-Michael
0.6	update system model, update matrix	Sophie
0.7	added UML	Jan-Michael
0.8	added User Interfaces	Jan-Michael
0.9	minor changes in matrix description update constraint simplification	Sophie
0.10	updated UML files	Jan-Michael
0.11	added Input Checker	Jan-Michael
0.12	added Command Line Interface	Jan-Michael
0.13	added description steps CS algorithm	Sophie
0.14	update command line description added Web Interface	Sophie
0.15	added GUI	Jan-Michael
1.0	first release	

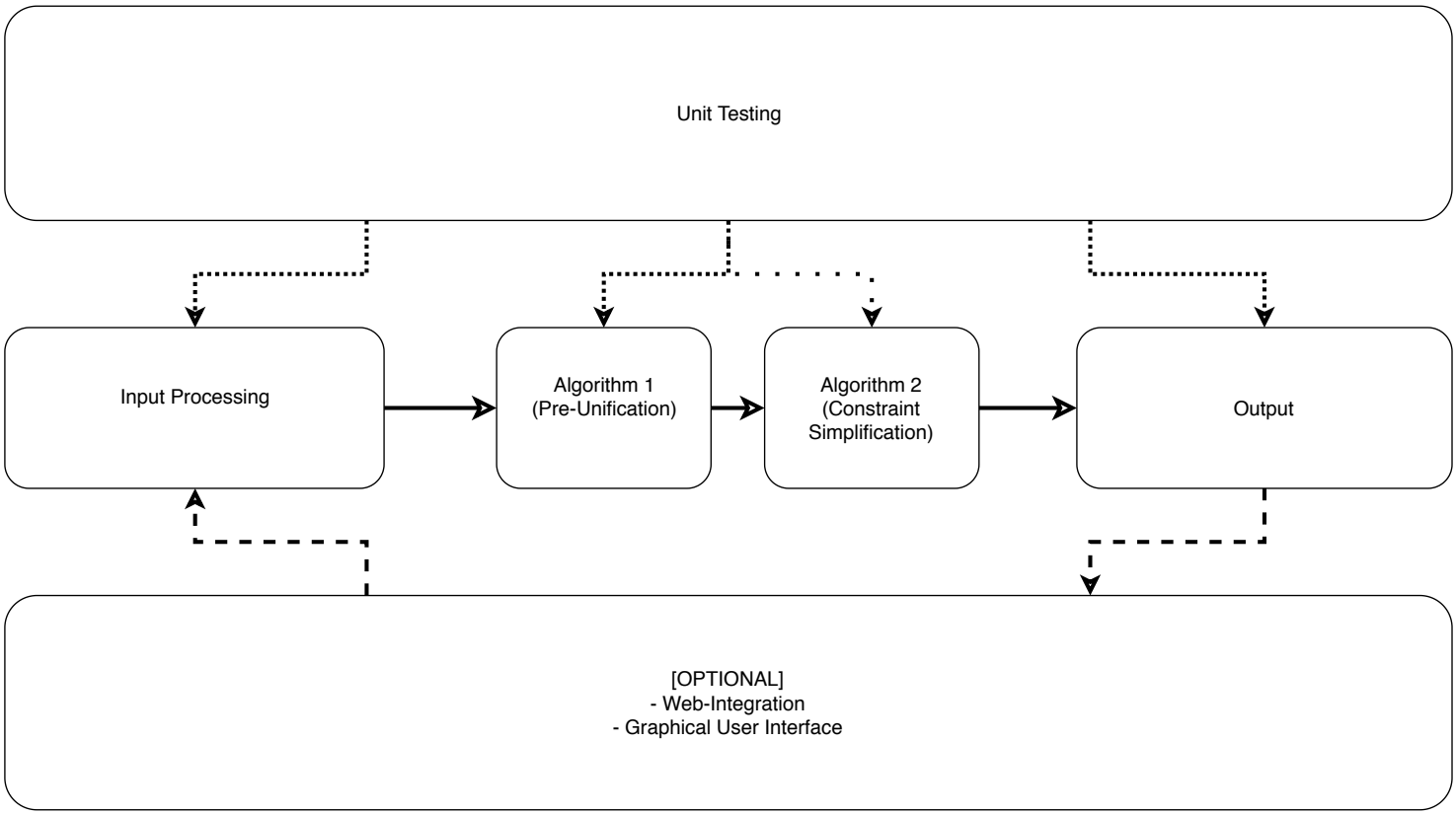
1 System Overview

We split the problem in 4 (5) smaller tasks:

1. Input Processing,
2. Pre-Unification,
3. Constraint Simplification,
4. Input/Output.

O. Web-Integration.

[*]jan.holzinger@gmx.at
[†]sophie@hofmanninger.co.at


Unit Testing
Input Processing
Algorithm 1
(Pre-Unification)
Algorithm 2
(Constraint
Simplification)
Output
[OPTIONAL]
- Web-Integration
- Graphical User Interface

1.1 Input Processing

1.1.1 InputParser

The first idea here is to copy, alter and extend the existing code, in the class InputParser.

1.1.2 Input Checker

We want to be able to check the input for example for missing parenthesis and so on. Therefore we will allow the user to specify an InputChecker. The idea is, to create an Interface, with just a method check(String):boolean. This then can be implemented by the implementation of the checkers.

1.1.3 Proximity Relations

For the Proximity Relations $\mathcal{R}$ and the *λ-cut* we have the following idea:

1. We try to get the number of function symbols with the same arity.
2. We let the user input the values to construct a symmetric matrix that consists of values in $[0, 1]$. This matrix must have a 1 in the main diagonal. All values below are 0. Therefore they will not be stored in the implementation.
3. We let the user (later) input $\lambda \in [0, 1]$ and calculate the set $\mathcal{R}_\lambda$.

The current implementation of InputParser creates a list of all Functions. This list is then sorted by arity, i.e. let f, g be functions with arity $a, b \in \mathbb{N}$ respectively. Then $a < b \Rightarrow f \prec g$.
Let now $n \in \mathbb{N}$ be the size of the list, the list then represents the caption of a $n \times n$ matrix. Assume the problem contains functions with arity 0,1,...,m. Let k_l be the number of Functions with arity l. Then

$$\sum_{l=0}^{m} k_l = n.$$

The matrix then looks like

$$\begin{array}{c} \\ f_{0_1} \\ f_{0_2} \\ \vdots \\ f_{0_{k_0}} \\ f_{1_1} \\ f_{1_2} \\ \vdots \\ f_{1_{k_1}} \\ \vdots \\ f_{m_1} \\ f_{m_2} \\ \vdots \\ f_{m_{k_m}} \end{array} \begin{array}{c} \begin{array}{cccccccccccccc} f_{0_1} & f_{0_2} & \cdots & f_{0_{k_0}} & f_{1_1} & f_{1_2} & \cdots & f_{1_{k_1}} & \cdots & f_{m_1} & f_{m_2} & \cdots & f_{m_{k_m}} \end{array} \\ \begin{pmatrix} 1 & (*) & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & 1 & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & \ddots & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & (*) & (*) & (*) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & * & 1 & (*) & (*) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & \ddots & (*) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & (*) & (*) & (*) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & 1 & (*) & (*) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & \ddots & (*) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & * & 1 \end{pmatrix} \end{array},$$

where:

- the 1's in the diagonal are fixed, as a function must have proximity 1 to itself,
- the 0's are fixed, as functions with different aritys can't be close,
- the values where the entry is * must be of the same value as their (*) counterpart, as the matrix is symmetric.

Hence, we only need to get the values for the positions marked with (*). We call such a position "open case", and we make a list of open cases. Before asking the user to enter relation values, we will generate the list of open cases inside of the matrix. After generation of the list, we will wait for user input. With the valid user input we will build the proximity relation matrix.

If we want to retrieve a value, i.e. if we want to know $\mathcal{R}(s_1, s_2)$ for given s_1, s_2, we can call $getRelation(s_1, s_2)$. To get all relation pairs of s_1, that have a relation value greater or equal to λ, the method $getRelations(s_1, \lambda)$ can be used. Moreover it is possible to get all reation elements of the matirix by calling $getListOfFunctions()$. This will return all $f_{i_{k_j}}$, where $0 \leq i \leq m$ and $0 \leq j \leq m$.

1.2 Algorithms

We implement the Algorithms in an own class, that has two public static functions, preUnification and constraintSimplification. The other methods are only used for the construction of the two simplification algorithms and therefore they are private and static. The methods should take two inputs, mandatory an unification problem, and optional a StringBuffer, to log certain events.

1.2.1 Pre-Unification Algorithm

The preUnification method consists of a loop, that runs until either $\mathrm{P} = \emptyset$ or it is detected, that there is no solution to the problem.
Inside the loop body, the 7 pre-unification rules are iteratively applied to the first element (which gets popped by doing so). The method returns true, iff the pre-unification was successful.

The method changes the problems constraints and pre-unifier accordingly.

1.2.2 Constraint Simplification Algorithm

The constraintSimplification method will be called, if the preUnification method returns true. As the preUnification method, the constraintSimplification method consists of a loop. This loop runs until the set of constraints is empty. As input the method needs the problem, more precisely the set of constraints, and the relation matrix $\mathcal{R}$. If the set of constraints could be simplified, the method returns true and otherwise false.
Inside the loop, the seven rules to simplify the constraint set, will be applied. Three of the seven rules are hidden, because the NN2 rule is integrated in the NN1 rule, the Fail1 is implemented in FFS and the Fail2 is integrated in the NN1 rule. Moreover it is possible to run the constraint simplification algorithm with and without logging.
The steps which are printed by logging can be read as followed: Every new line symbolises a branch. In front of every row the branch-number is printed. As soon as a new branch will be created there will appear *(Create Branch a - b)*, where *a* and *b* denote the branch-numbers. For example if branch 5 and 6 will be created, it looks like *(Create Branch 5 - 6)*. Moreover it is possible that a branch will be branched. This follows the same notation and can be imagined as a tree.

1.3 Input/Output

We provide a command line user interface and a web application for the program.

1.3.1 Command line interface

The command line user interface is implemented in the "SPC_CL" class(SolvingProximityConstraints_via_CommandLine). The program can be accessed with no or some of this optional arguments: Note: the first argument if any are given is always the equation in the format "lhs =? rhs" or "lhs = rhs".

the equation in the format "lhs =? rhs" (left hand side and right hand side of the equation.

-f <PATH> a path given to a file where (parts) of the proximity relations are stored the format must follow the convention, that is mentioned below.

-s silent mode on/off, default is off, -s with no argument switches on, optional argument “n” switches off.

-l followed by the lambda value.

The proximity relations must be stored in a text file, one line for each relation, where the two functions (or constants) are separated with “,” (comma) and the lambda value, also separated with “,”.

1.3.2 Web interface

For the web interface we provide a second project which is identical to the original, but it is a WEB-project. In our case the WEB-project is only thought to run the web application. To run the web application, it is important, that the project is a WEB-project, all java files are in the classes folder (WEB-INF) and the tomcat server is correctly adjusted.

To compile the web application the user has to open the jsp file, which is placed in the WebContent folder, in Eclipse. After pressing the run button, the console asks, if it is allowed to start the server. Press okay and the web interface appears.

It can be entered more than one problem by splitting them with a “;”. But be aware, that only the first one will be solved. Note: As in the command line interface the equation should be in the format “lhs =? rhs” or “lhs = rhs”.

You can choose the mode silent or detailed, depending on whether you want to see the steps or not. Afterwards you can set a lambda. Pressing *Reset* resets the input and you can start always from the beginning. Pressing *Enter Function* creates a new table, where the number of additional function can the entered for the relation matrix. For example, if $q(a, b) = p(x, y)$ and a is also related to c then we have to enter 1 additional function with arity 0. By pressing *Enter additional Functions* the user can now build the relation matrix. Pressing *Enter Matrix* yields the solution of the problem.

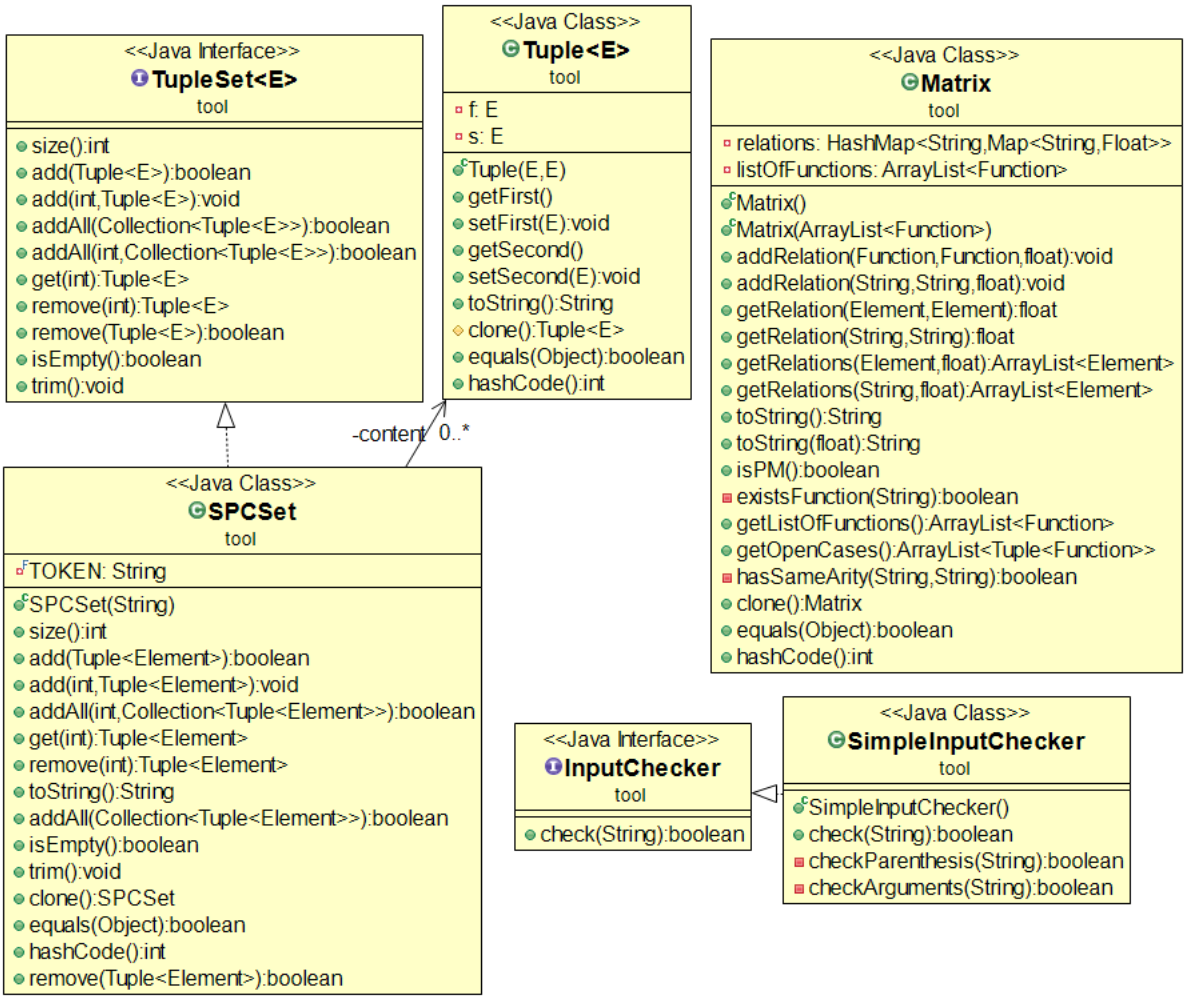
1.3.3 Graphical User Interface

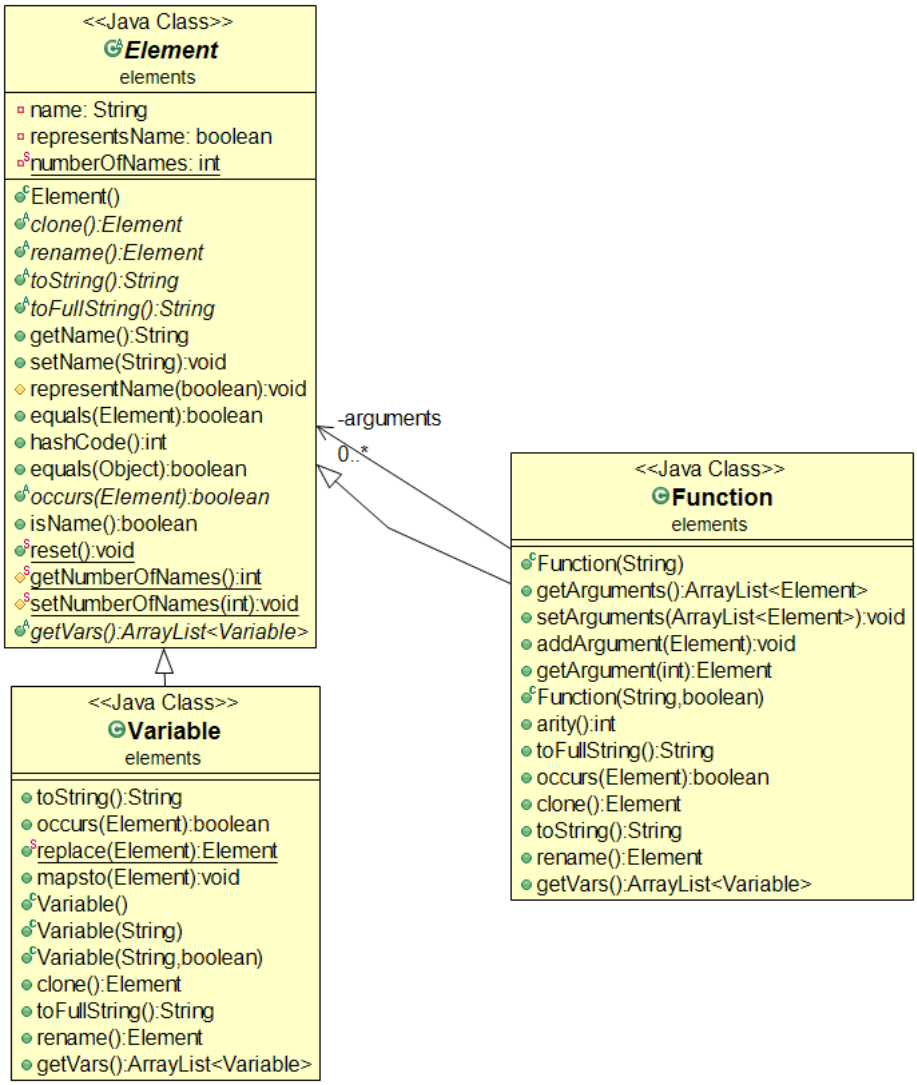
The GUI constists of parts: Equation, Constraints and Results. The user first enters the equation, then the open cases are calculated and shown. Then the user can enter the corresponding proximity relations and click unify. Then the result is shown. If wanted, the second step (constraint simplification) can be performed again - with different lambda.

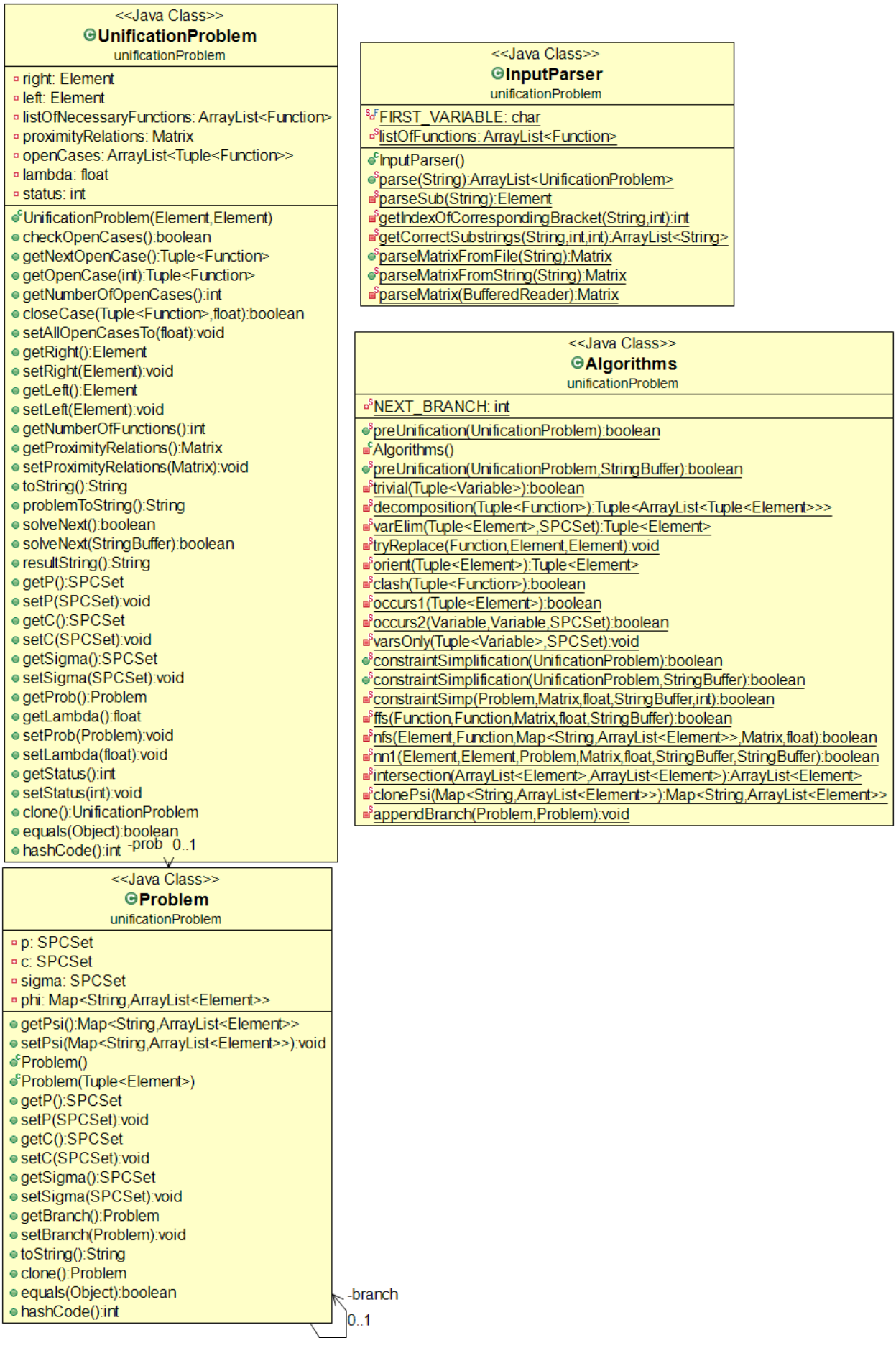
2 System Model

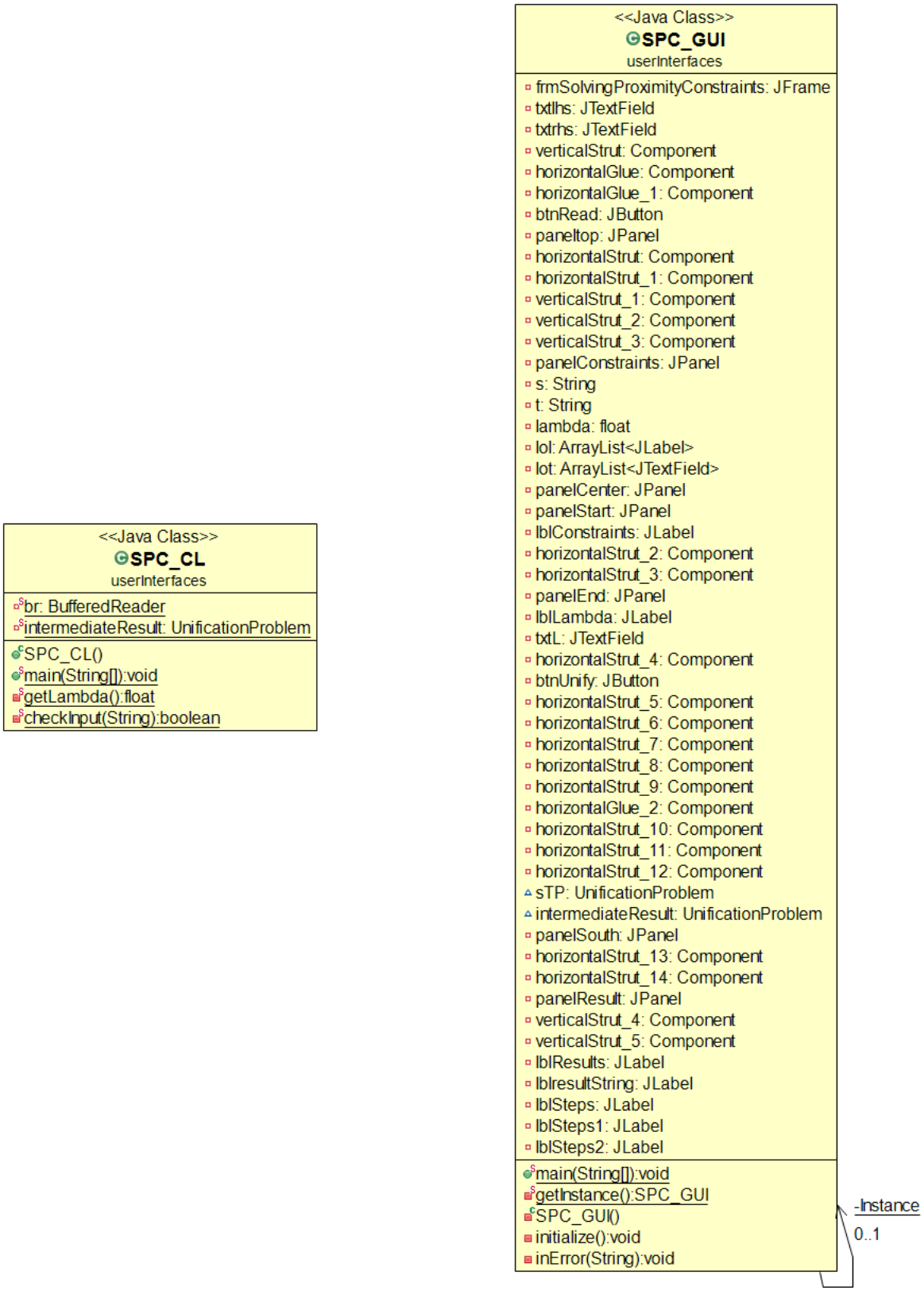
The program consists of 4 packages,

- tool
- elements
- unificationProblem
- userInterfaces


<<Java Interface>>
TupleSet<E>
tool
size():int
add(Tuple<E>):boolean
add(int,Tuple<E>):void
addAll(Collection<Tuple<E>>):boolean
addAll(int,Collection<Tuple<E>>):boolean
get(int):Tuple<E>
remove(int):Tuple<E>
remove(Tuple<E>):boolean
isEmpty():boolean
trim():void
<<Java Class>>
Tuple<E>
tool
f: E
s: E
Tuple(E,E)
getFirst()
setFirst(E):void
getSecond()
setSecond(E):void
toString():String
clone():Tuple<E>
equals(Object):boolean
hashCode():int
<<Java Class>>
Matrix
tool
relations: HashMap<String,Map<String,Float>>
listOfFunctions: ArrayList<Function>
Matrix()
Matrix(ArrayList<Function>)
addRelation(Function,Function,float):void
addRelation(String,String,float):void
getRelation(Element,Element):float
getRelation(String,String):float
getRelations(Element,float):ArrayList<Element>
getRelations(String,float):ArrayList<Element>
toString():String
toString(float):String
isPM():boolean
existsFunction(String):boolean
getListOfFunctions():ArrayList<Function>
getOpenCases():ArrayList<Tuple<Function>>
hasSameArity(String,String):boolean
clone():Matrix
equals(Object):boolean
hashCode():int
-content
0..*
<<Java Class>>
SPCSet
tool
TOKEN: String
SPCSet(String)
size():int
add(Tuple<Element>):boolean
add(int,Tuple<Element>):void
addAll(int,Collection<Tuple<Element>>):boolean
get(int):Tuple<Element>
remove(int):Tuple<Element>
toString():String
addAll(Collection<Tuple<Element>>):boolean
isEmpty():boolean
trim():void
clone():SPCSet
equals(Object):boolean
hashCode():int
remove(Tuple<Element>):boolean
<<Java Interface>>
InputChecker
tool
check(String):boolean
<<Java Class>>
SimpleInputChecker
tool
SimpleInputChecker()
check(String):boolean
checkParenthesis(String):boolean
checkArguments(String):boolean

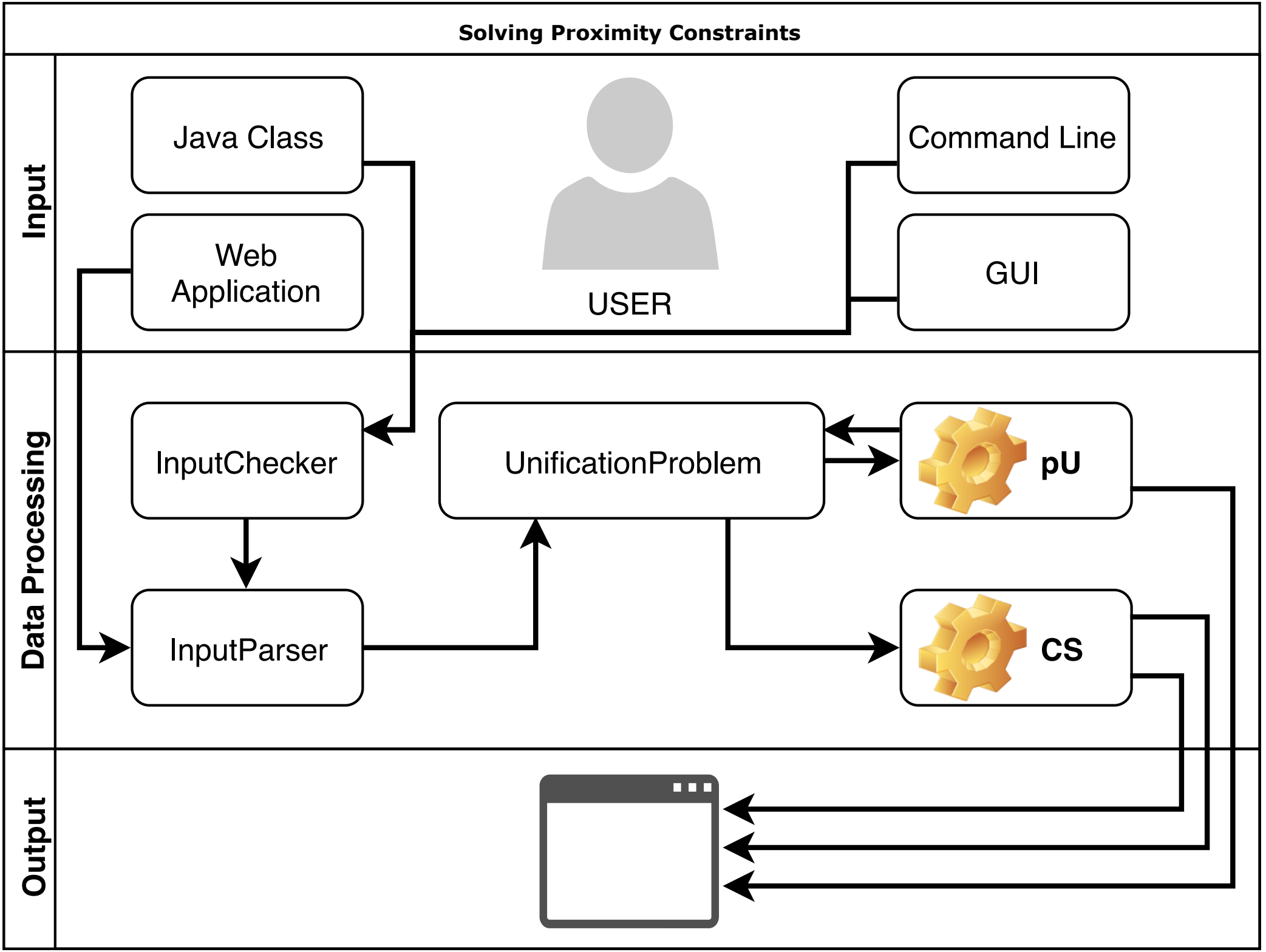

<<Java Class>>
Element
elements
name: String
representsName: boolean
numberOfNames: int
Element()
clone():Element
rename():Element
toString():String
toFullString():String
getName():String
setName(String):void
representName(boolean):void
equals(Element):boolean
hashCode():int
equals(Object):boolean
occurs(Element):boolean
isName():boolean
reset():void
getNumberOfNames():int
setNumberOfNames(int):void
getVars():ArrayList<Variable>
-arguments
0..*
<<Java Class>>
Function
elements
Function(String)
getArguments():ArrayList<Element>
setArguments(ArrayList<Element>):void
addArgument(Element):void
getArgument(int):Element
Function(String,boolean)
arity():int
toFullString():String
occurs(Element):boolean
clone():Element
toString():String
rename():Element
getVars():ArrayList<Variable>
<<Java Class>>
Variable
elements
toString():String
occurs(Element):boolean
replace(Element):Element
mapsto(Element):void
Variable()
Variable(String)
Variable(String,boolean)
clone():Element
toFullString():String
rename():Element
getVars():ArrayList<Variable>


<<Java Class>>
UnificationProblem
unificationProblem
right: Element
left: Element
listOfNecessaryFunctions: ArrayList<Function>
proximityRelations: Matrix
openCases: ArrayList<Tuple<Function>>
lambda: float
status: int
UnificationProblem(Element,Element)
checkOpenCases():boolean
getNextOpenCase():Tuple<Function>
getOpenCase(int):Tuple<Function>
getNumberOfOpenCases():int
closeCase(Tuple<Function>,float):boolean
setAllOpenCasesTo(float):void
getRight():Element
setRight(Element):void
getLeft():Element
setLeft(Element):void
getNumberOfFunctions():int
getProximityRelations():Matrix
setProximityRelations(Matrix):void
toString():String
problemToString():String
solveNext():boolean
solveNext(StringBuffer):boolean
resultString():String
getP():SPCSet
setP(SPCSet):void
getC():SPCSet
setC(SPCSet):void
getSigma():SPCSet
setSigma(SPCSet):void
getProb():Problem
getLambda():float
setProb(Problem):void
setLambda(float):void
getStatus():int
setStatus(int):void
clone():UnificationProblem
equals(Object):boolean
hashCode():int
-prob
0..1
<<Java Class>>
InputParser
unificationProblem
FIRST_VARIABLE: char
listOfFunctions: ArrayList<Function>
InputParser()
parse(String):ArrayList<UnificationProblem>
parseSub(String):Element
getIndexOfCorrespondingBracket(String,int):int
getCorrectSubstrings(String,int,int):ArrayList<String>
parseMatrixFromFile(String):Matrix
parseMatrixFromString(String):Matrix
parseMatrix(BufferedReader):Matrix
<<Java Class>>
Algorithms
unificationProblem
NEXT_BRANCH: int
preUnification(UnificationProblem):boolean
Algorithms()
preUnification(UnificationProblem,StringBuffer):boolean
trivial(Tuple<Variable>):boolean
decomposition(Tuple<Function>):Tuple<ArrayList<Tuple<Element>>>
varElim(Tuple<Element>,SPCSet):Tuple<Element>
tryReplace(Function,Element,Element):void
orient(Tuple<Element>):Tuple<Element>
clash(Tuple<Function>):boolean
occurs1(Tuple<Element>):boolean
occurs2(Variable,Variable,SPCSet):boolean
varsOnly(Tuple<Variable>,SPCSet):void
constraintSimplification(UnificationProblem):boolean
constraintSimplification(UnificationProblem,StringBuffer):boolean
constraintSimp(Problem,Matrix,float,StringBuffer,int):boolean
ffs(Function,Function,Matrix,float,StringBuffer):boolean
nfs(Element,Function,Map<String,ArrayList<Element>>,Matrix,float):boolean
nn1(Element,Element,Problem,Matrix,float,StringBuffer,StringBuffer):boolean
intersection(ArrayList<Element>,ArrayList<Element>):ArrayList<Element>
clonePsi(Map<String,ArrayList<Element>>):Map<String,ArrayList<Element>>
appendBranch(Problem,Problem):void
<<Java Class>>
Problem
unificationProblem
p: SPCSet
c: SPCSet
sigma: SPCSet
phi: Map<String,ArrayList<Element>>
getPsi():Map<String,ArrayList<Element>>
setPsi(Map<String,ArrayList<Element>>):void
Problem()
Problem(Tuple<Element>)
getP():SPCSet
setP(SPCSet):void
getC():SPCSet
setC(SPCSet):void
getSigma():SPCSet
setSigma(SPCSet):void
getBranch():Problem
setBranch(Problem):void
toString():String
clone():Problem
equals(Object):boolean
hashCode():int
-branch
0..1



3 Work Flow

The typical workflow looks like this:


Solving Proximity Constraints
Input
Java Class
Web Application
USER
Command Line
GUI
Data Processing
InputChecker
InputParser
UnificationProblem
pU
CS
Output