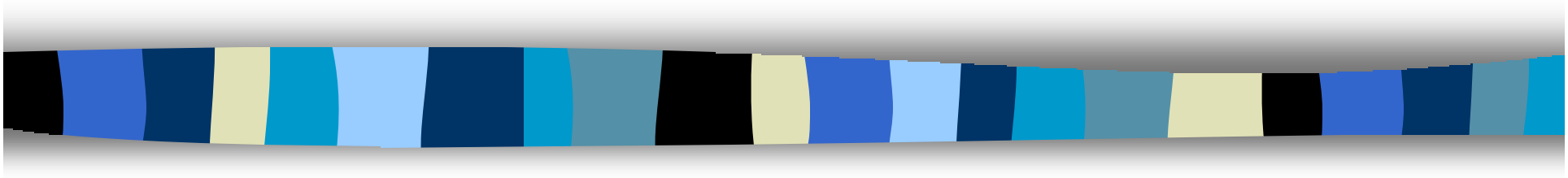
Implantation de PtidejSolver en Java



Salim Bensemmane

Iyadh Sidhom

Fayçal Skhiri

Département d'informatique et de recherche opérationnelle

Université de Montréal

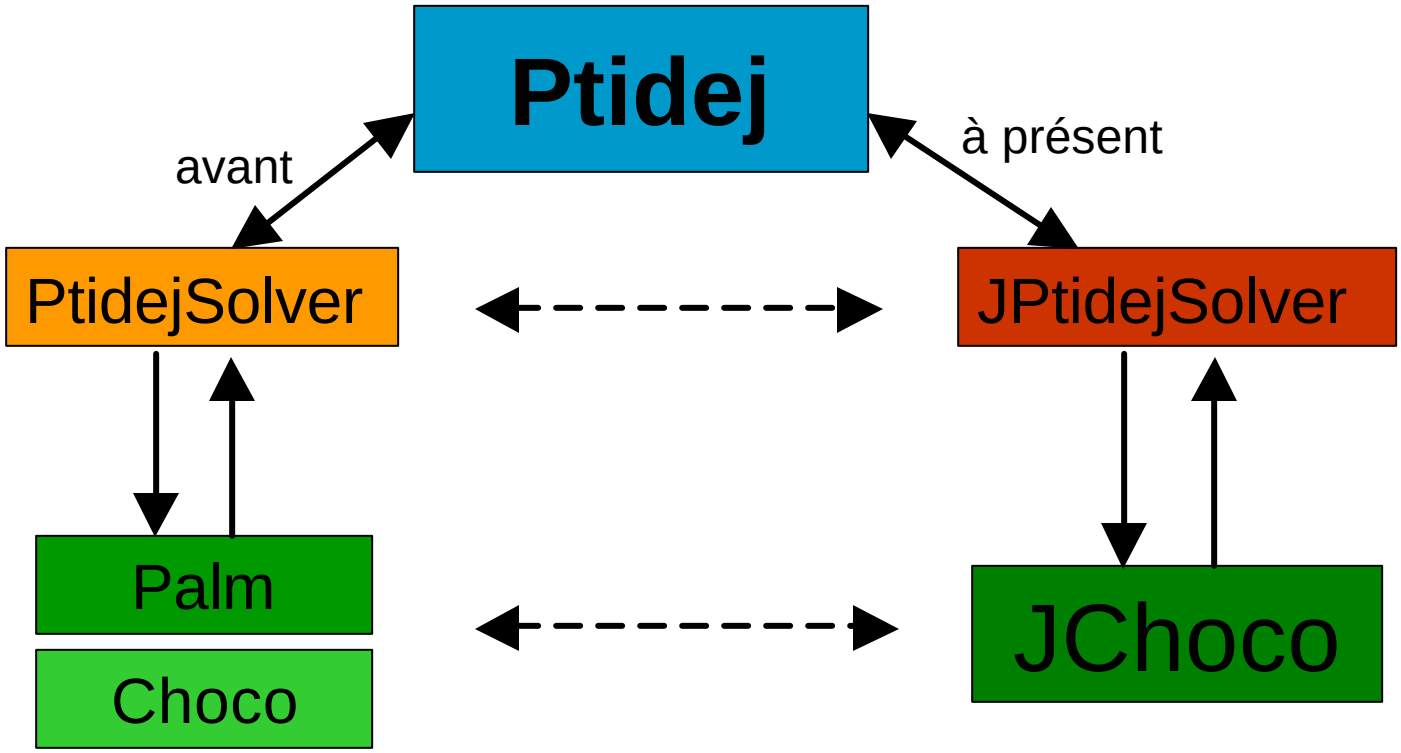
Sommaire

- Présentation du projet
- Architecture JPtidejSolver
- Concepts des différents solveurs
- Architecture des contraintes
- Critiques & Observations
- Évolutions personnels
- Exemple d’application

Présentation du projet

- Objectif
  - Travail de retro-conception
  - Sachant que Ptidej utilise le module PtidejSolver (en Claire) qui fait appel aux bibliothèques Palm & Choco
  - Notre tâche consiste à implanter un module équivalent JPtidejSolver (en Java) qui utilise la bibliothèque Jchoco (équivalente à Palm & Choco)

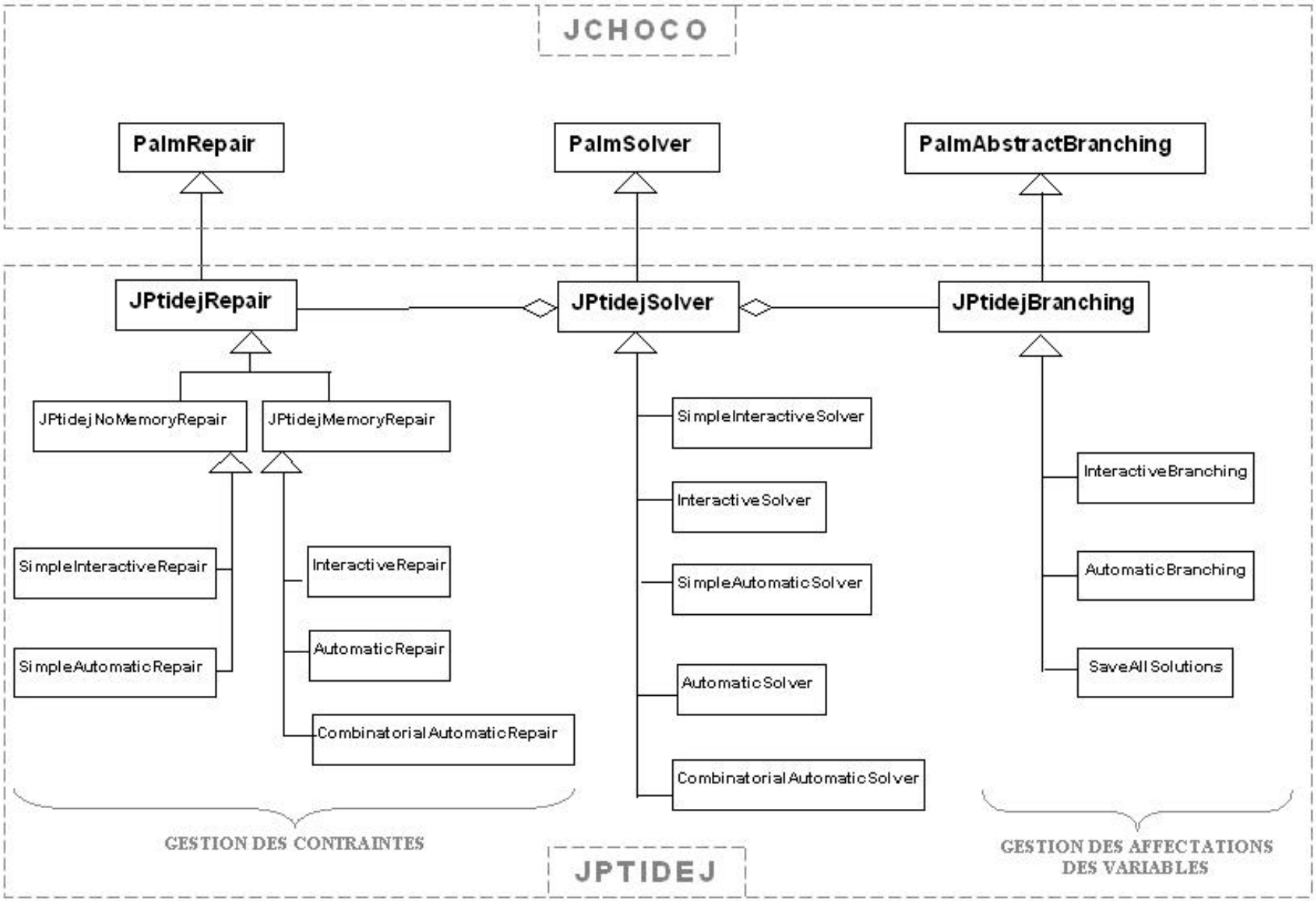
Présentation du projet (suite)



But de PtidejSolver

- PtidejSolver est un module qui permet de déterminer le patron de conception utilisé par un programme écrit en Java en le comparant avec un modèle décrit sous la forme d'un système de contraintes

Architecture JPtidejSolver



Architecture JPtidejSolver (suite)

- Caractéristiques
  - Architecture structurée
- Architecture extensible : possibilité de rajouter d'autres solveurs, avec leurs repairs et branching correspondants

SimpleInteractiveSolver

- Ce solveur permet à l'usager d'interagir à travers la console
- Les contraintes qui posent contradiction sont identifiées et affichées
- L'usager sélectionne une contrainte afin d'être relaxée
- Nous propageons le problème de nouveau

InteractiveSolver

- Ce solveur interagit avec l'usager de la même manière que SimpleInteractiveSolver
- La seule différence est lorsque nous relaxons une contrainte nous la remplaçons par la suivante dans sa famille (StrictInheritance $\rightarrow$ Inheritance $\rightarrow$ StrictInheritancePath $\rightarrow$ InheritancePath)
- Dans l'étape suivante nous proposons à l'usager de remettre une contrainte parmi celles qui ont été relaxées auparavant

SimpleAutomaticSolver

- Ce solveur retire les contraintes automatiquement
- Pas d'interaction avec l'usager
- Nous lançons la résolution avec le solveur de JChoco
- En cas de contradiction, nous retirons la contrainte relaxable de plus faible poids puis nous propageons les contraintes de nouveau

AutomaticSolver

- Même principe que le simpleAutomaticSolver
- Mais en plus il se caractérise par
  - Sa mémorisation des contraintes à retirer
- Une fois que toutes les contraintes sont relaxées, nous ajoutons de nouveau les contraintes retirées

CombinatorialSolver

- Définition de tous les problèmes possibles à partir du problème de base
- Soit un problème P avec N contraintes relaxables
  - Génération des catégories de problèmes
    - Catégorie 0 : aucune contrainte relaxable n'est relaxée
    - Catégorie 1 : 1 contrainte relaxable est relaxée
    - …
    - Catégorie N : toutes les contraintes relaxables sont relaxées

CombinatorialSolver (suite)

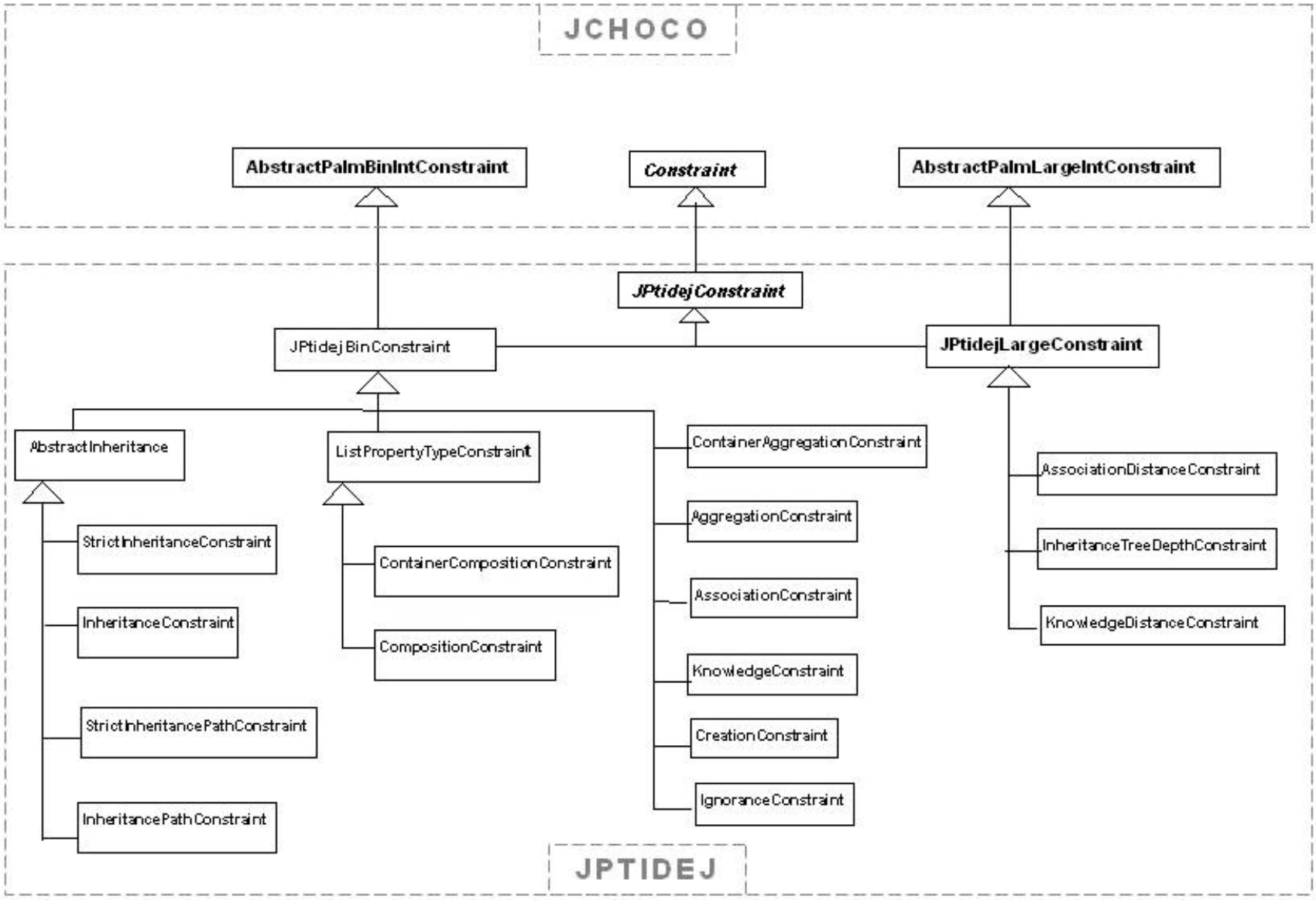
- Pour chaque catégorie, soit la « catégorie p » nous déterminons l'ensemble des problèmes possibles (de cardinalité C^p_n)

- De façon indépendante, nous lançons la résolution (avec le solveur de JChoco) pour chaque problème P_i appartenant à la « catégorie p »

CombinatorialSolver (suite)

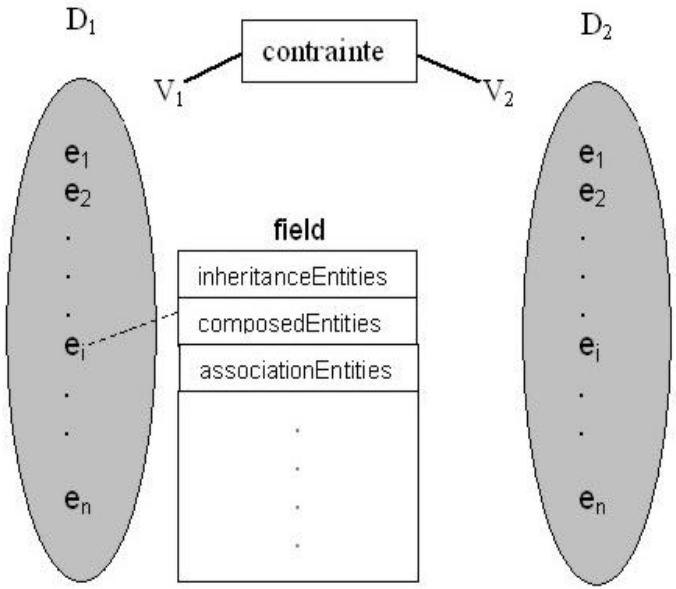
- Dans le processus de résolution, dès qu'une contrainte relaxable de moindre poids est détectée, elle est systématiquement remplacée par sa contrainte suivante dans la même famille, puis nous relançons à nouveau le processus de résolution pour le problème dérivé

Architecture des contraintes



Architecture des contraintes (suite)

- Optimisation structurelle des contraintes
  - Basée sur le caractère générique des entités du domaine des variables
  - Correspond à la factorisation des méthodes de nettoyage (propagate et awakeOnRemove)



Critique & Observations

- Difficulté de trouver une documentation sur le langage « Claire » du faite que ce langage est peu utilisé
- Fort couplage de la bibliothèque JChoco
- Non respect des normes de programmation en Java (ex. : des variables d’instance publiques !)

Critique & Observations (suite)

- Le plan de travail a été sujet à des changements suite aux difficultés de départ mentionnées précédemment
- Malgré les difficultés, les délais ont été respectés
- Le plus
  - Dans le rapport (section VIII), nous avons fourni une liste de bogues (conceptuels et logiques) avec leurs corrections

Critique & Observations (suite)

- L’architecture de JPtidejSolver est structurée et extensible (possibilité de rajouter des solveurs et des contraintes)
- Optimisation structurelle de l’architecture des contraintes (factorisation des méthodes de nettoyage)

Évolution personnelle

- Ce projet était intéressant pour plusieurs raisons
  - Il nous a permis de nous familiariser avec l'environnement de développement Eclipse très utilisé dans l'industrie
  - Nous avons acquis de l'expérience sur l'outil de travail en équipe CVS qui a été très utile pour rendre le code disponible à tous les membres de l'équipe

Évolution personnelle (suite)

- Ce projet nous a permit d'approfondir notre connaissance sur certains aspects de la programmation orientée objets en Java tels que : la réflexion, le polymorphisme et l'héritage…

- Aussi nous avons appris un nouveau concept de programmation qui est la programmation par contraintes

Exemple d’application

```
package ptidej.solver.test;

import ptidej.solver.JPtidejProblem;
import ptidej.solver.JPtidejVar;
import ptidej.solver.constraint.StrictInheritanceConstraint;
import choco.integer.IntVar;

public class AtomaticSolverTest
{

public static void main(String[] args)
{
JPtidejProblem pb = new JPtidejProblem(90, " test5 ", "test4.txt");
final int SUP = pb.getAllEntities().size() - 1;
final int INF = 0;
JPtidejVar v1 = new JPtidejVar(pb, "v1", IntVar.BOUNDS, INF, SUP);
JPtidejVar v2 = new JPtidejVar(pb, "v2", IntVar.BOUNDS, INF, SUP);
JPtidejVar v3 = new JPtidejVar(pb, "v3", IntVar.BOUNDS, INF, SUP);

pb.addVar(v1);
pb.addVar(v2);
pb.addVar(v3);
```

Exemple d’application (suite)

```
StrictInheritanceConstraint c1 =
new StrictInheritanceConstraint(
"StrictInheritanceConstraint v1 inherits from v2",
"command",
v1,
v2,
100);
StrictInheritanceConstraint c2 =
new StrictInheritanceConstraint(
"StrictInheritanceConstraint v2 inherits from v3",
"command",
v2,
v3,
60);
StrictInheritanceConstraint c3 =
new StrictInheritanceConstraint(
"StrictInheritanceConstraint v3 inherits from v2",
"command",
v3,
v2,
80);

pb.post(c1);
pb.post(c2);
pb.post(c3);

pb.automaticSolve(true,"automaticResult");
}
}
```

Résultat

Solutions without constraint:
  # -|>-: StrictInheritanceConstraint v2 inherits from v3

1.84.Name = test5
1.84.XCommand = command
1.84.v1 = A
1.84.v2 = B
1.84.v3 = A

…

Solutions without constraint:
  # -|>-: StrictInheritanceConstraint v2 inherits from v3

3.84.Name = test5
3.84.XCommand = command
3.84.v1 = C
3.84.v2 = D
3.84.v3 = C

Solutions without constraints:
 # added constraint: -|>-: StrictInheritanceConstraint v2 inherits from v3
  # -|>-: StrictInheritanceConstraint v3 inherits from v2

4.67.Name = test5
4.67.XCommand = command
4.67.v1 = A
4.67.v2 = B
4.67.v3 = A

…