Contributo alla Specifica JSR-331 mediante un'Implementazione basata su JSetL

Candidato: **Fabio Biselli**
Relatore: **Prof. Federico Bergenti**

Università degli Studi di Parma

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,
   - allocazione delle risorse,

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,
   - allocazione delle risorse,
   - configurazione, e altri.

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,
   - allocazione delle risorse,
   - configurazione, e altri.
2. Necessità di applicazioni per CSP.

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,
   - allocazione delle risorse,
   - configurazione, e altri.
2. Necessità di applicazioni per CSP.
3. Java nell'industria software.

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,
   - allocazione delle risorse,
   - configurazione, e altri.
2. Necessità di applicazioni per CSP.
3. Java nell'industria software.
4. Solver di CSP scritti in Java:

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,
   - allocazione delle risorse,
   - configurazione, e altri.
2. Necessità di applicazioni per CSP.
3. Java nell'industria software.
4. Solver di CSP scritti in Java:
   - CHOCO, JaCoP, Constrainer.

Introduzione

CSP e Java in ambito professionale

1. CSP nell'industria:
   - pianificazione,
   - allocazione delle risorse,
   - configurazione, e altri.
2. Necessità di applicazioni per CSP.
3. Java nell'industria software.
4. Solver di CSP scritti in Java:
   - CHOCO, JaCoP, Constrainer.
5. Una sola API per diversi solver.

API javax

Cosa sono.

Le API javax sono un insieme di interfacce standard disponibili al programmatore che estendono il linguaggio Java.

API javax

Cosa sono.

Le API javax sono un insieme di interfacce standard disponibili al programmatore che estendono il linguaggio Java.

Chi le definisce.

Il Java Community Process (JCP) è l'istituzione che si occupa di regolare lo sviluppo della tecnologia Java.

API javax

Cosa sono.

Le API javax sono un insieme di interfacce standard disponibili al programmatore che estendono il linguaggio Java.

Chi le definisce.

Il Java Community Process (JCP) è l'istituzione che si occupa di regolare lo sviluppo della tecnologia Java.

Chi può proporle.

Ogni membro del JCP può fare richiesta di aggiornamento, creazione o modifica di una nuova specifica. Questa viene denominata Java Specification Request (JSR).

JSR-331

Cos'è?

JSR-331 è una *richiesta per specifiche* Java. Queste specifiche definiscono le API per la programmazione a vincoli.

JSR-331

Cos'è?

JSR-331 è una *richiesta per specifiche* Java. Queste specifiche definiscono le API per la programmazione a vincoli.

A chi è rivolta?

La specifica è rivolta principalmente ad aziende che utilizzano CSP, ai fornitori di risolutori di vincoli e a ricercatori in ambito di programmazione con vincoli.

JSR-331

Cos'è?

JSR-331 è una *richiesta per specifiche* Java. Queste specifiche definiscono le API per la programmazione a vincoli.

A chi è rivolta?

La specifica è rivolta principalmente ad aziende che utilizzano CSP, ai fornitori di risolutori di vincoli e a ricercatori in ambito di programmazione con vincoli.

Chi partecipa?

Al JSR-331 partecipa un gruppo di esperti, il responsabile del processo di specifica è il Dr. Jacob Feldman dell'Università di Cork.

JSR-331

Iter del processo

L'approvazione di un JSR passa attraverso ad un iter istituzionale.

JSR-331

Iter del processo

L'approvazione di un JSR passa attraverso ad un iter istituzionale.

Le tappe del JSR-331

JSR-331

Iter del processo

L'approvazione di un JSR passa attraverso ad un iter istituzionale.

Le tappe del JSR-331

- 17 Agosto 2009, la proposta di standardizzazione viene accettata dal JCP.
- 25 Marzo 2010, il primo progetto di valutazione viene pubblicato.
- 19 Agosto 2011, la proposta finale viene sottoposta al JCP.
- 20 Febbraio 2012, JSR-331 viene approvato.

JSR-331

Iter del processo

L'approvazione di un JSR passa attraverso ad un iter istituzionale.

Le tappe del JSR-331

- 17 Agosto 2009, la proposta di standardizzazione viene accettata dal JCP.
- 25 Marzo 2010, il primo progetto di valutazione viene pubblicato.
- 19 Agosto 2011, la proposta finale viene sottoposta al JCP.
- 20 Febbraio 2012, JSR-331 viene approvato.

Inizio della collaborazione

Con l'inizio del lavoro di tirocinio e tesi inizia il progetto di implementazione mediante JSetL, nei primi di Novembre viene fornito dal Dr. Feldman uno spazio dedicato a JSetL nel repository di JSR-331.

JSR-331

Struttura

JSR-331 prescrive un insieme di operazioni fondamentali, la struttura consiste in tre principali componenti.

JSR-331

Struttura

JSR-331 prescrive un insieme di operazioni fondamentali, la struttura consiste in tre principali componenti.

- Specifiche (CP API).
- Implementazione basata su differenti CP solver.
- *TCK* (*Tecnology Compatibility Kit*), un pacchetto di test, utilizzato per verificare la conformità alle specifiche delle varie implementazioni.

JSR-331

Specifiche

Le specifiche per JSR-331 consistono in:

JSR-331

Specifiche

Le specifiche per JSR-331 consistono in:

Un'interfaccia pura

Il package `javax.constraints`, contiene i maggiori concetti e metodi per definire e risolvere CSP.

JSR-331

Specifiche

Le specifiche per JSR-331 consistono in:

Un'interfaccia pura

Il package `javax.constraints`, contiene i maggiori concetti e metodi per definire e risolvere CSP.

La *common implementation*

Il package `javax.constraints.impl`, contiene definizioni parziali o totali di oggetti, concetti di risoluzione e metodi che non dipendono direttamente da uno specifico CP solver.

JSR-331

Implementazioni

Ogni implementazione del JSR-331 è basata su uno specifico solver e deve definire tutte le interfacce. I package richiesti dalla specifica:

JSR-331

Implementazioni

Ogni implementazione del JSR-331 è basata su uno specifico solver e deve definire tutte le interfacce. I package richiesti dalla specifica:

Definizione del problema

Il package `javax.constraints.impl` deve fornire classi Java come `Problem`, `Constraints` o `Var`.

JSR-331

Implementazioni

Ogni implementazione del JSR-331 è basata su uno specifico solver e deve definire tutte le interfacce. I package richiesti dalla specifica:

Definizione del problema

Il package `javax.constraints.impl` deve fornire classi Java come `Problem`, `Constraints` o `Var`.

Vincoli

Il package `javax.constraints.impl.constraint` deve fornire vincoli come `Or`, `IfThen` o `AllDifferent`.

JSR-331

Implementazioni

Ogni implementazione del JSR-331 è basata su uno specifico solver e deve definire tutte le interfacce. I package richiesti dalla specifica:

Definizione del problema

Il package `javax.constraints.impl` deve fornire classi Java come `Problem`, `Constraints` o `Var`.

Vincoli

Il package `javax.constraints.impl.constraint` deve fornire vincoli come `Or`, `IfThen` o `AllDifferent`.

Risoluzione

Il package `javax.constraints.impl.search` deve fornire classi come `Solver` o `SearchStrategy`.

JSR-331

Technology Compatibility Kit

Il *TCK* è un pacchetto di documentazione, test e strumenti utilizzati per testare la correttezza delle implementazioni.

JSR-331

Technology Compatibility Kit

Il *TCK* è un pacchetto di documentazione, test e strumenti utilizzati per testare la correttezza delle implementazioni.

Consiste in due package:

JSR-331

Technology Compatibility Kit

Il *TCK* è un pacchetto di documentazione, test e strumenti utilizzati per testare la correttezza delle implementazioni.

Consiste in due package:

Tests (`org.jcp.jsr331.tests`)

Contiene i moduli JUnit che permettono di validare automaticamente la correttezza dell'implementazione.

JSR-331

Technology Compatibility Kit

Il *TCK* è un pacchetto di documentazione, test e strumenti utilizzati per testare la correttezza delle implementazioni.

Consiste in due package:

Tests (`org.jcp.jsr331.tests`)

Contiene i moduli JUnit che permettono di validare automaticamente la correttezza dell'implementazione.

Samples (`org.jcp.jsr331.samples`)

Contiene esempi di CSP che forniscono test integrati per i più comuni vincoli e strategie di ricerca incluse in JSR-331.

Implementazione

Analisi

Studio della specifica

Mediante il documento di specifica pubblicato e le interfacce fornite.

Implementazione

Analisi

Studio della specifica

Mediante il documento di specifica pubblicato e le interfacce fornite.

Valutazione delle funzionalità di JSetL

Mediante la documentazione disponibile ed i sorgenti.

Implementazione

Analisi

Studio della specifica

Mediante il documento di specifica pubblicato e le interfacce fornite.

Valutazione delle funzionalità di JSetL

Mediante la documentazione disponibile ed i sorgenti.

Relazioni tra JSR-331 e JSetL

Associazione dei requisiti della specifica ed il solver JSetL.

Implementazione

Analisi

Studio della specifica

Mediante il documento di specifica pubblicato e le interfacce fornite.

Valutazione delle funzionalità di JSetL

Mediante la documentazione disponibile ed i sorgenti.

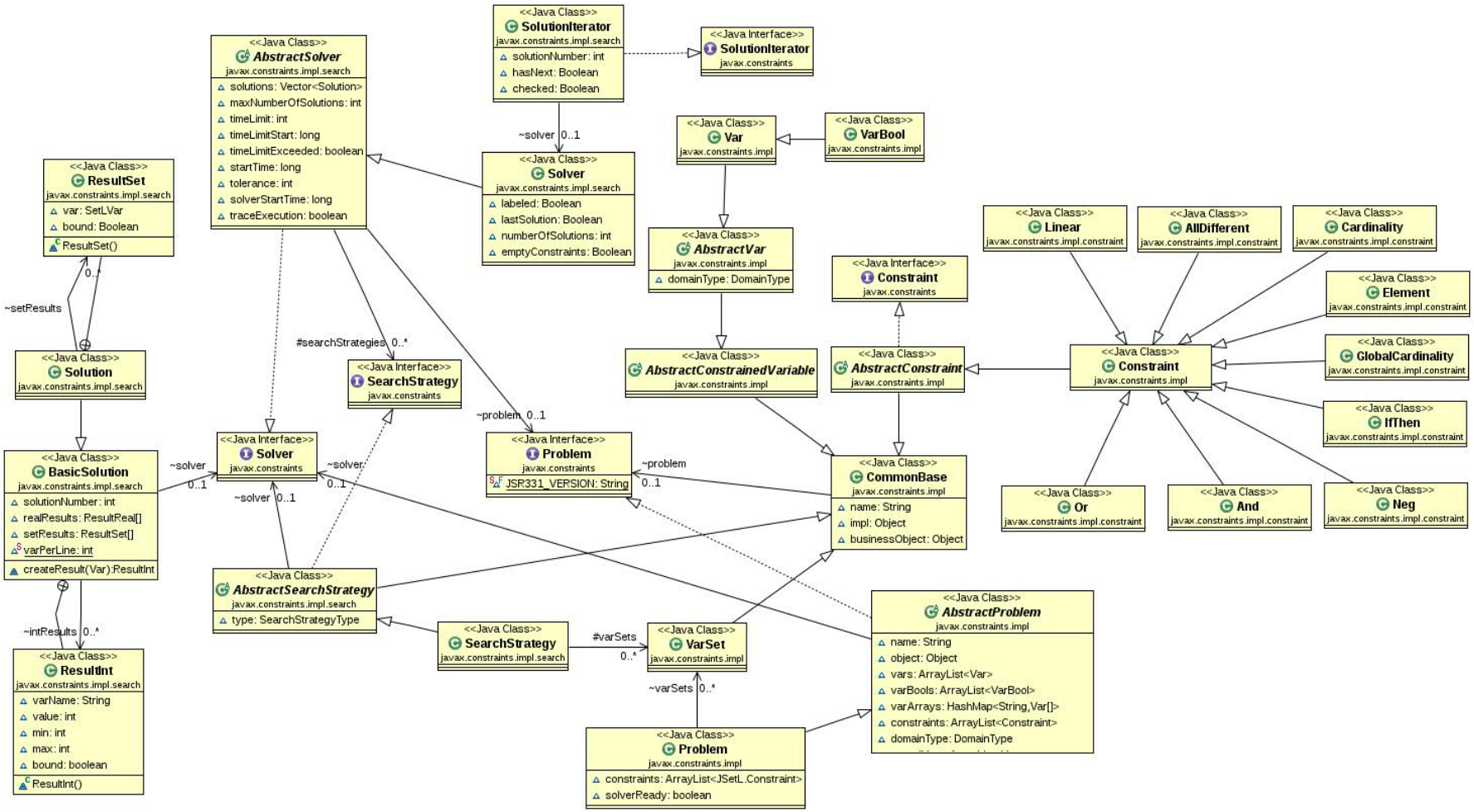
Relazioni tra JSR-331 e JSetL

Associazione dei requisiti della specifica ed il solver JSetL.

Sono risultati utili esempi nel package sample del TCK.

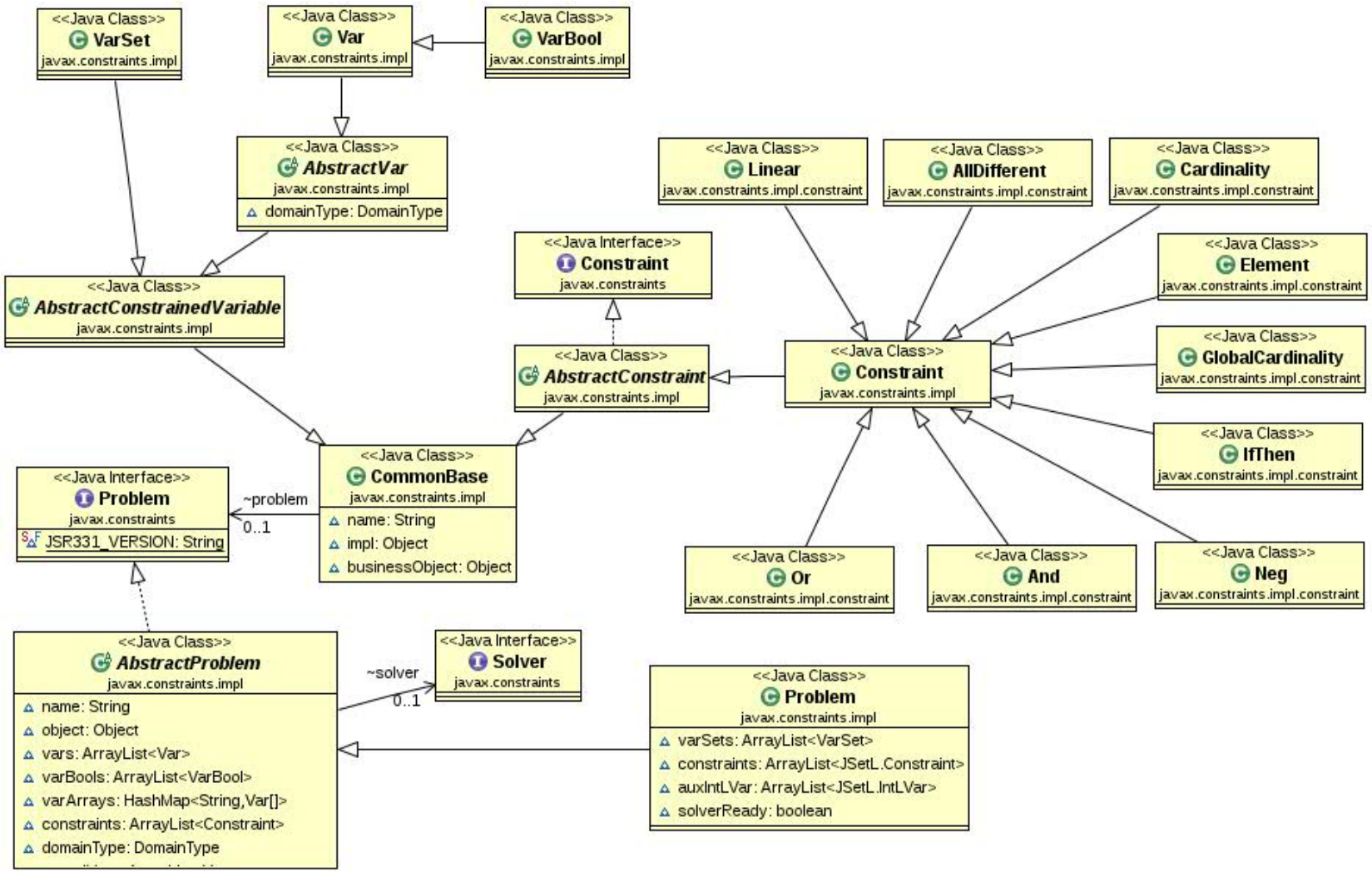
Implementazione

Class Diagram



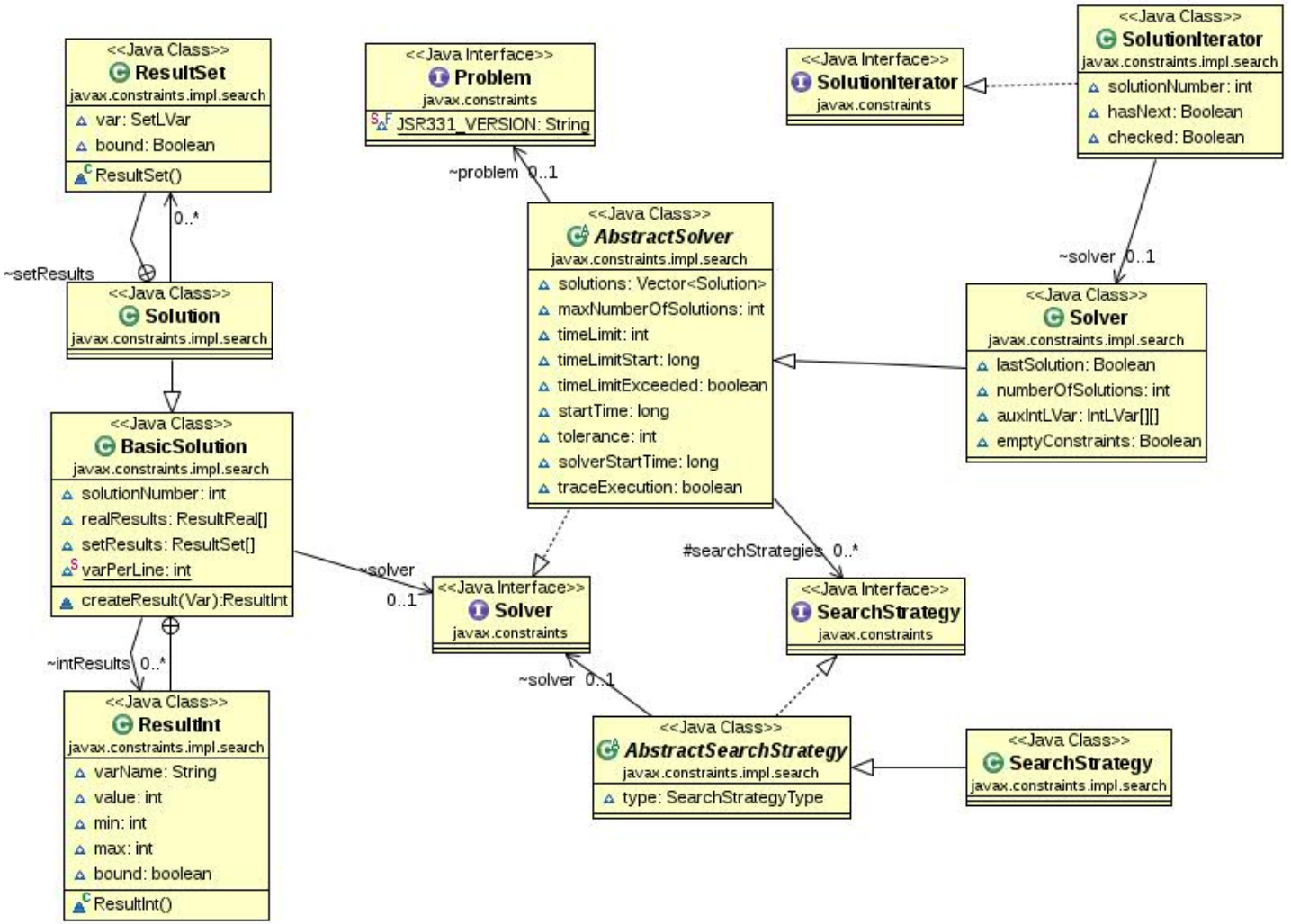
Implementazione

Class Diagram: definizione del problema



Implementazione

Class Diagram: soluzione del problema



Implementazione

Codifica

Nella fase di codifica si sono implementate 18 classi suddivise tra i tre package richiesti:

Implementazione

Codifica

Nella fase di codifica si sono implementate 18 classi suddivise tra i tre package richiesti:

Definizione del problema

`Problem`, `Constraint`, `Var`, `VarBool` e `VarSet`.

Implementazione

Codifica

Nella fase di codifica si sono implementate 18 classi suddivise tra i tre package richiesti:

Definizione del problema

`Problem`, `Constraint`, `Var`, `VarBool` e `VarSet`.

Vincoli

`AllDifferent`, `Cardinality`, `Element`, `GlobalCardinality`, `Linear`, `And`, `IfThen`, `Neg` e `Or`.

Implementazione

Codifica

Nella fase di codifica si sono implementate 18 classi suddivise tra i tre package richiesti:

Definizione del problema

`Problem`, `Constraint`, `Var`, `VarBool` e `VarSet`.

Vincoli

`AllDifferent`, `Cardinality`, `Element`, `GlobalCardinality`, `Linear`, `And`, `IfThen`, `Neg` e `Or`.

Risoluzione

`Solver`, `SearchStrategy`, `Solution` e `SolutionIterator`.

Implementazione

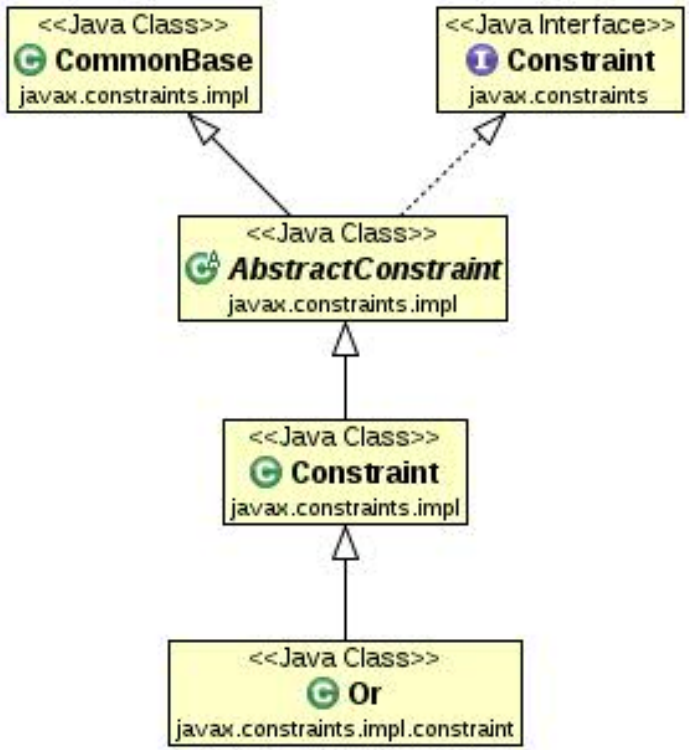
Mapping: disgiunzione

Gerarchia

Sia `Or` che `Constraint` estendono la classe astratta fornita da JSR-331, che implementa l'interfaccia e si basa su `CommonBase`.

`CommonBase`

È una classe contenitore, utile alle implementazioni poiché fornisce servizi *getter* e *setter*.



Implementazione

Mapping: disgiunzione

Dati due vincoli `c1` e `c2`, la classe `Or` mediante il costruttore genera un nuovo vincolo `c` tale che:

$$c = c1 \vee c2$$

```
public Or(Constraint c1, Constraint c2) {
        super(c1.getProblem());
        Constraint result = (Constraint) c1.or(c2);
        setImpl(result.getImpl());
}
```

Implementazione

Mapping: disgiunzione

Dati due vincoli c1 e c2, la classe `Or` mediante il costruttore genera un nuovo vincolo c tale che:

$$c = c1 \vee c2$$

```
public Or(Constraint c1, Constraint c2) {
        super(c1.getProblem());
        Constraint result = (Constraint) c1.or(c2);
        setImpl(result.getImpl());
}
```

Il metodo `or(Constraint)` della classe `Constraint`.

```
public Constraint or(javax.constraints.Constraint x) {
        JSetL.Constraint c1 = this.getConstraint();
        JSetL.Constraint c2 = ((Constraint) x).getConstraint();
        Constraint c = new Constraint(getProblem(), c1.or(c2));
        return c;
}
```

Implementazione

La classe `Solver`

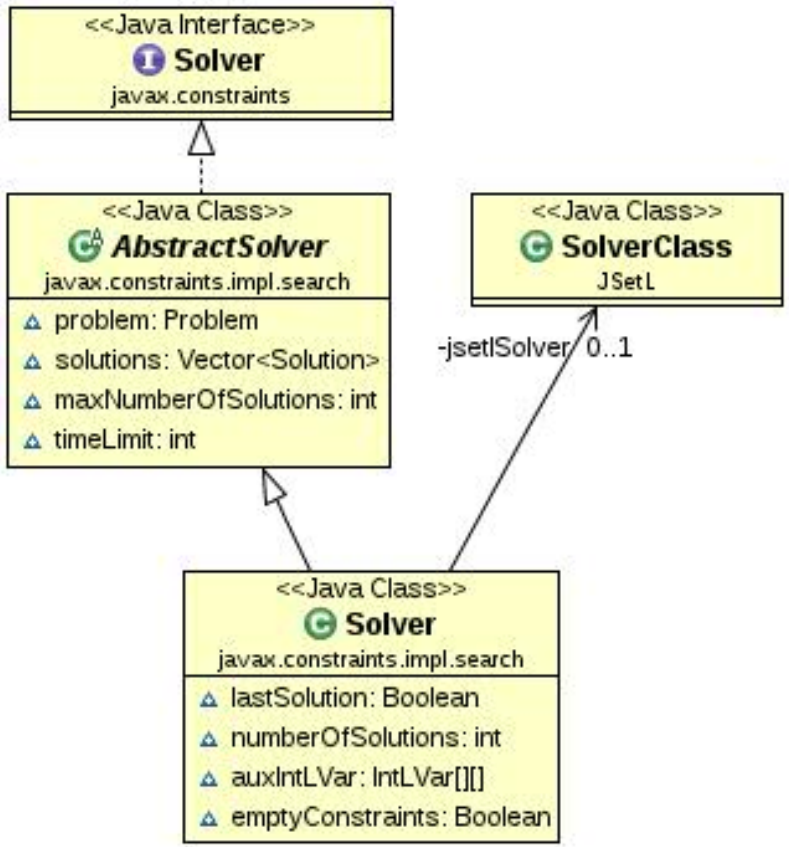
Gerarchia

Estende `AbstractSolver` che non sfrutta la classe `CommonBase`, non è quindi prevista un'implementazione specifica.

```
public class Solver extends AbstractSolver {

  private SolverClass jsetlSolver;
  .
  .
```

Il solver JSetL è inserito tra gli attributi della classe.



Implementazione

La classe `Solver`

Costruttore

Il costruttore della classe istanzia il solver JSetL, quindi vi inserisce tutti i vincoli salvati dal problema.

```
public Solver(Problem problem) {
  super(problem);
  jsetlSolver = new SolverClass()
      ;
  numberOfSolutions = 0;
  getProblemConstraints();
  setProblemConstraints();
  getAuxVariables();
}
```

Implementazione

La classe `Solver`

Costruttore

Il costruttore della classe istanzia il solver JSetL, quindi vi inserisce tutti i vincoli salvati dal problema.

```
public Solver(Problem problem) {
  super(problem);
  jsetlSolver = new SolverClass()
        ;
  numberOfSolutions = 0;
  getProblemConstraints();
  setProblemConstraints();
  getAuxVariables();
}
```

Ricerca di una soluzione

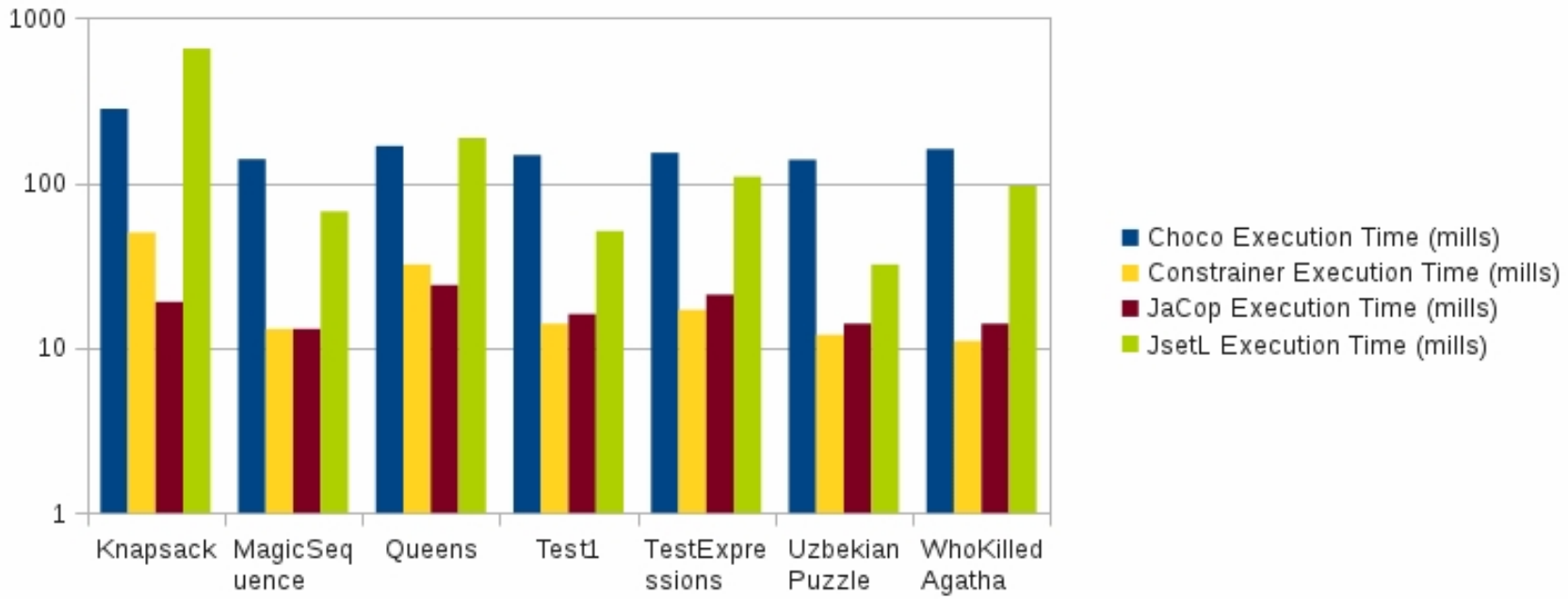
Il cuore del metodo `findSolution`, invoca la `solve` di JSetL e quindi crea la soluzione con il relativo costruttore.

```
public Solution findSolution(
      ProblemState restoreOrNot) {
  .
  .
  jsetlSolver.solve();
  solution = new Solution(this,
                numberOfSolutions++);
  .
  .
```

Valutazione delle prestazioni

Test semplici

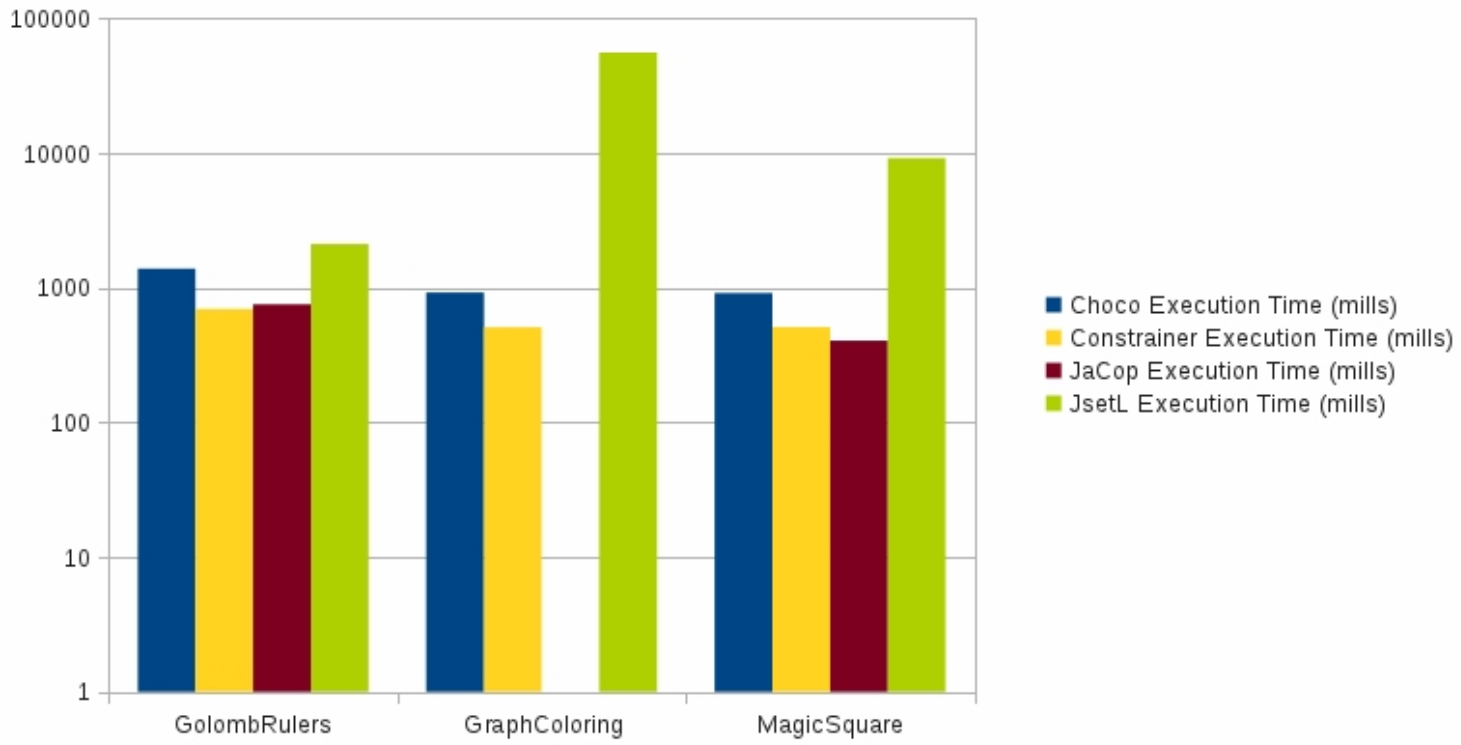
Test semplici



Valutazione delle prestazioni

Test complessi

Test complessi



Conclusioni

Il lavoro di tesi verte sui seguenti punti fondamentali:

Conclusioni

Il lavoro di tesi verte sui seguenti punti fondamentali:

- Studio della specifica JSR-331.
- Studio del solver JSetL.
- Realizzazione dell'implementazione.
- Test di validazione e coverage.
- Valutazioni delle prestazioni.
- Comparazione con altri solver.

JSR-331 ha fornito un ottimo banco di prova per JSetL, nonché l'opportunità di migliorarne alcuni aspetti.

Lavori futuri

Si evidenziano i seguenti possibili sviluppi futuri:

Lavori futuri

Si evidenziano i seguenti possibili sviluppi futuri:

- Estendere l'implementazione con funzionalità specifiche di JSetL non previste dal JSR-331.
- Mantenere allineata l'implementazione con gli sviluppi futuri della specifica.
- Valutare implementazioni alternative per la specifica.
- Integrare nuove funzionalità offerte da JSetL.
- Contribuire allo sviluppo del TCK.
- Pubblicazione di un rapporto tecnico (in fase di stesura).

Lavori futuri

Si evidenziano i seguenti possibili sviluppi futuri:

- Estendere l'implementazione con funzionalità specifiche di JSetL non previste dal JSR-331.
- Mantenere allineata l'implementazione con gli sviluppi futuri della specifica.
- Valutare implementazioni alternative per la specifica.
- Integrare nuove funzionalità offerte da JSetL.
- Contribuire allo sviluppo del TCK.
- Pubblicazione di un rapporto tecnico (in fase di stesura).

Grazie!

Esempi

Un esempio di CSP

```
Problem p = new Problem("TestXYZ");
Var x = p.variable("X", 0, 10);
Var y = p.variable("Y", 0, 10);
Var z = p.variable("Z", 0, 10);
p.post(x,"<",y);  // X < Y
p.post(x.plus(y),"=",z);  // X + Y = Z
p.post(y,">",5);
Var cost = p.variable("Cost", 2, 25);
// Cost = 3XY - 4Z
p.post(cost,"=",x.multiply(3).multiply(y).minus(z.multiply(4)));
return p;

Problem problem = defineCsp();
problem.log("=== Optimal Solution:");
Solver solver = problem.getSolver();
Var costVar = problem.getVar("Cost");
Solution solution = solver.findOptimalSolution(Objective.MAXIMIZE, costVar);
if (solution == null)
        problem.log("No Solutions");
else
        solution.log();
problem.log("Cost=" + solution.getValue("Cost"));
assertTrue(solution.getValue("Cost") == 23);
```