L'énoncé de l'examen de Janvier 2019

Le patron Specification : Fowler et Evans

&

VIP

Variability and Injection Pattern

language and framework

jean-michel Douin, douin au cnam point fr

version : 14 Janvier 2020

Bibliographie principale utilisée

- **Specifications, M.Fowler & E. Evans**
  - https://martinfowler.com/apsupp/spec.pdf
- **Rule Object, Ali Arsanjani**
  - A Pattern Language for Adaptive and ScalableBusiness Rule Construction
    https://hillside.net/plop/plop2k/proceedings/Arsanjani/Arsanjani.pdf
- **NFP121 Enoncé de l'examen de Janvier 2019**
  - http://jfod.cnam.fr/NFP121/annales/2019_janvier/
- **http://blog.xebia.fr, utilisation du patron Specification, Nicolas Lecoz**
  - https://blog.xebia.fr/2009/12/29/le-pattern-specification-pour-la-gestion-de-vos-regles-metier/

Sommaire

- **L'énoncé de l'examen de janvier 2019**
  - **Fortement inspiré de Specification de Fowler & Evans**

- **La proposition femtoContainer**
  - **VIP pour Variability and Injection Pattern**

Prémisses

- ***Hypothèses***
  - ***Les règles métiers changent régulièrement.***

- ***Objectifs***
  - *Comment prendre en compte ces changements tout en maintenant les systèmes maintenables, réutilisables et extensibles ?*
  - *Comment représenter des règles, pour une plus grande réutilisation et assurer une maintenabilité plus facile ?*

- ***Constat***
  - *Ces règles sont généralement implémentées dans les méthodes d'un objet métier. Elles font référence à d'autres objets métier, et prennent souvent la forme d'instructions* ***"si-alors-sinon",*** *disséminées dans le code devenant de fait, difficile à faire évoluer, à maintenir...*

Exemple possible de dissémination

- **Le calcul des congés,**
  - **Un exemple réel aperçu dans un cours précédent,**
  - **Règles de calcul selon la loi,**
  - **Ces règles peuvent changer,**
  - **Elles peuvent être différentes selon le lieu**
    - **Alsace-Moselle, Agent ultramarin, corse, conjoint ultramarin, CDD, CDI, en Apprentissage, congés pris avant le mois de mai, journée du Maire, …etc.**
  - **Il est tentant(fréquent) d'installer ces règles dans le code…**
  - **A chaque évolution, se reporter dans le code et modifier par l'ajout d'une instruction de type**
    - **if condition alors commande**

Le patron Specification

- **Selon M. Fowler et E. Evans**
- **Regrouper toutes les règles « au même endroit »**
- **Un arbre, comme une succession de règles à exécuter**
  - **Si arbre alors patron composite**
- **Règles:**
  - **Si Conditions1 alors Commandes1**
  - **Si Conditions2 alors Commandes2**
  - **Si Conditions3 alors Commandes3**
  - **…**
  - **Règle, Condition et Commandes sont des « instances du patron Composite »**

Une Règle est de la forme

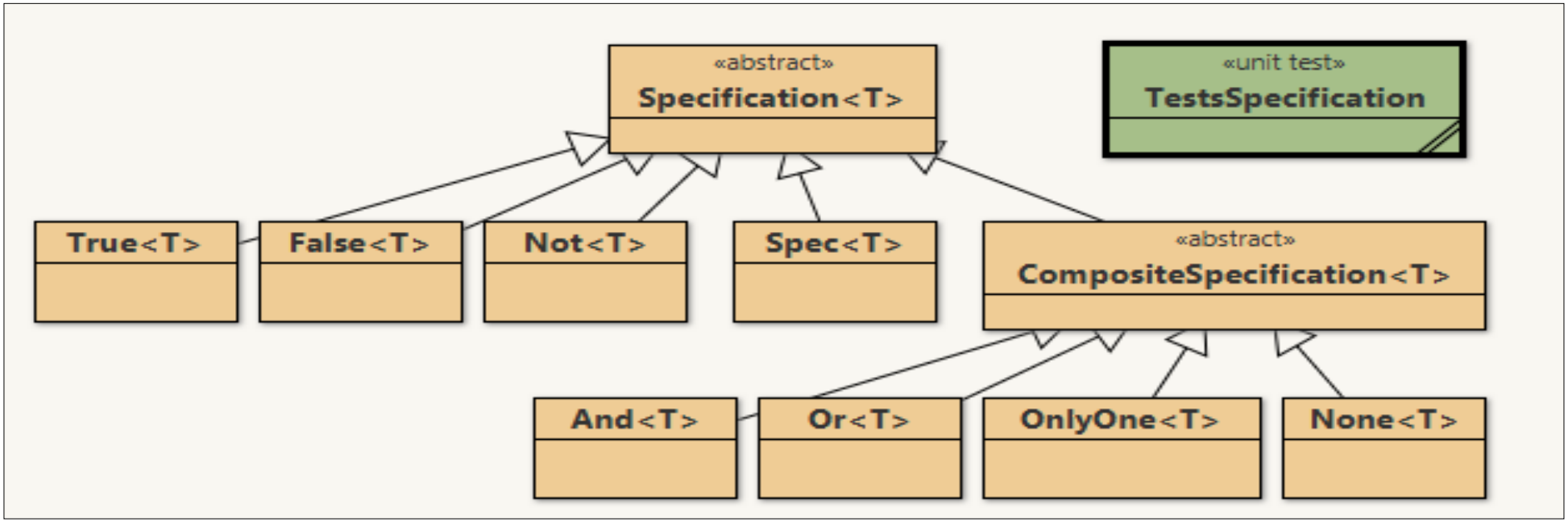
Si *la spécification est satisfaite*

alors *la commande* est exécutée.

- Les patrons ***Specification***, ***Command*** et ***Rule*** permettent de définir une suite de règles de type ***si-alors-sinon***
- **Le patron *Specification* permet de représenter la ou les conditions à satisfaire,**
- **Le patron *Command* représente la ou les commandes à exécuter,**
- **Enfin, le patron *Rule* exprime une règle ou une suite de règles de type:**

Si la spécification1 est satisfaite alors la commande1 est exécutée.
Si la spécification2 est satisfaite alors la commande2 est exécutée.
Si la …

Patron Specification : un composite




```
/** La classe abstraite Specification.
 * @param <T> L'entité métier, sur laquelle porte la condition.
 */
public abstract class Specification<T>{

    /** La condition à satisfaire.
     * @param t le paramètre générique de la méthode
     */
    public abstract boolean isSatisfiedBy(T t);

}
```

True, False, Not…

```
public class True<T> extends Specification<T>{
 public boolean isSatisfiedBy(T t){
     return true;
    }
 }
}

public class False<T> extends Specification<T>{
 public boolean isSatisfiedBy(T t){
     return false;
    }
 }
}

public class Not<T> extends Specification<T>{
    protected Specification<T> spec;

    public Not(Specification<T> spec){
        this.spec = spec;
    }

    public boolean isSatisfiedBy(T t){
        return !spec.isSatisfiedBy(t);
    }
}
```

Un exemple (très) simple

- **L'entité métier un entier … Integer… est pair comme condition**

```
class EstPair extends Specification<Integer>{
        public boolean isSatisfiedBy(Integer i){
            return i%2==0;
        }
}
```

- **Usage**

```
Specification<Integer> pair = new EstPair();
assertTrue(pair.isSatisfiedBy(4));
assertFalse(pair.isSatisfiedBy(5));
```

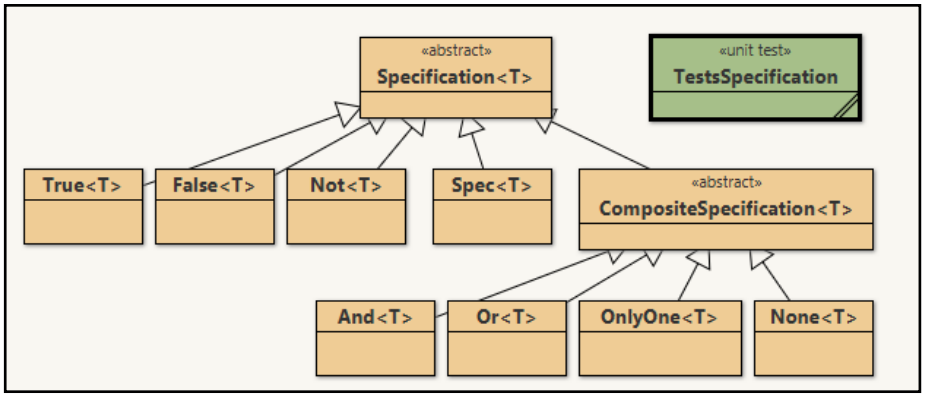
Un exemple simple, suite

```
class EstInferieur extends Specification<Integer>{
        private int valeur;
        public EstInferieur(int valeur){ this.valeur = valeur; }

        public boolean isSatisfiedBy(Integer i){
            return i < valeur;
        }
}
```

- **Usage**

```
Specification<Integer> inf = new EstInferieur(10);
assertTrue(inf.isSatisfiedBy(4));
assertFalse(inf.isSatisfiedBy(15));
```

Le patron Composite Specification




```
abstract class CompositeSpecification<T>
   extends Specification<T>
  implements Iterable<Specification<T>>{

    protected List<Specification<T>> children;

    public CompositeSpecification(){
        this.children = new ArrayList<>();
    }

    public CompositeSpecification<T> add(Specification<T> spec){
        this.children.add(spec);
        return this;
    }
    public Iterator<Specification<T>> iterator(){
        return this.children.iterator();
    }
 }
```

And, Or …

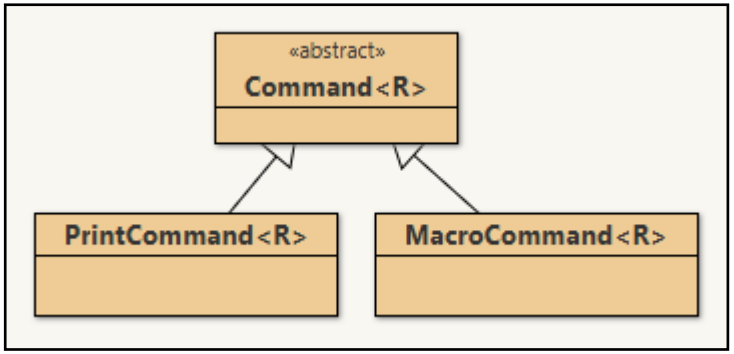
```
public class And<T> extends CompositeSpecification<T>{
    public boolean isSatisfiedBy(T t){
        boolean res = true;
        for(Specification<T> spec : this){
            res = res && spec.isSatisfiedBy(t); // And
          // res = res || spec.isSatisfiedBy(t); // Or
        }
        return res;
    }

}
```

Un exemple simple, suite

- **Usage**

```
Specification<Integer> pair = new EstPair();
Specification<Integer> inf = new EstInferieur(10);

Specification<Integer> and = new And(pair,inf);

Specification<Integer> or = new Or(pair,and);

assertTrue(and.isSatisfiedBy(4));
assertFalse(or.isSatisfiedBy(5));
```

Commande




```
public abstract class Command<R>{
    /** L'exécution de la commande
     * @param r l'entité transmise
     * @return le résultat retourné
     */
    public abstract R execute(R r) throws Exception;
```

- **Objectif, rappel :**
- **Si *la spécification est satisfaite* alors *la commande* est exécutée.**

Un exemple simple

```
public static class Inc extends Command<Integer>{
        public Integer execute(Integer i){
            return new Integer(i+1);
        }
}
```

- **Usage**

```
Command<Integer> inc = new Inc();
assertEquals(3, inc.execute(2).intValue());

Integer res = inc.execute(4);
```

MacroCommande, le composite

```
public class MacroCommand<R> extends Command<R>
                             implements Iterable<Command<R>>{
    private List<Command<R>> commands;

    public MacroCommand(){this.commands = new ArrayList<>(); }

    public MacroCommand<R> add(Command<R> command){
        commands.add(command);
        return this;
    }

    public R execute(R r) throws Exception{
        for(Command<R> cmd : this){
            r = cmd.execute(r);
        }
        return r;
    }

    public Iterator<Command<R>> iterator(){
        return commands.iterator();
    }
```

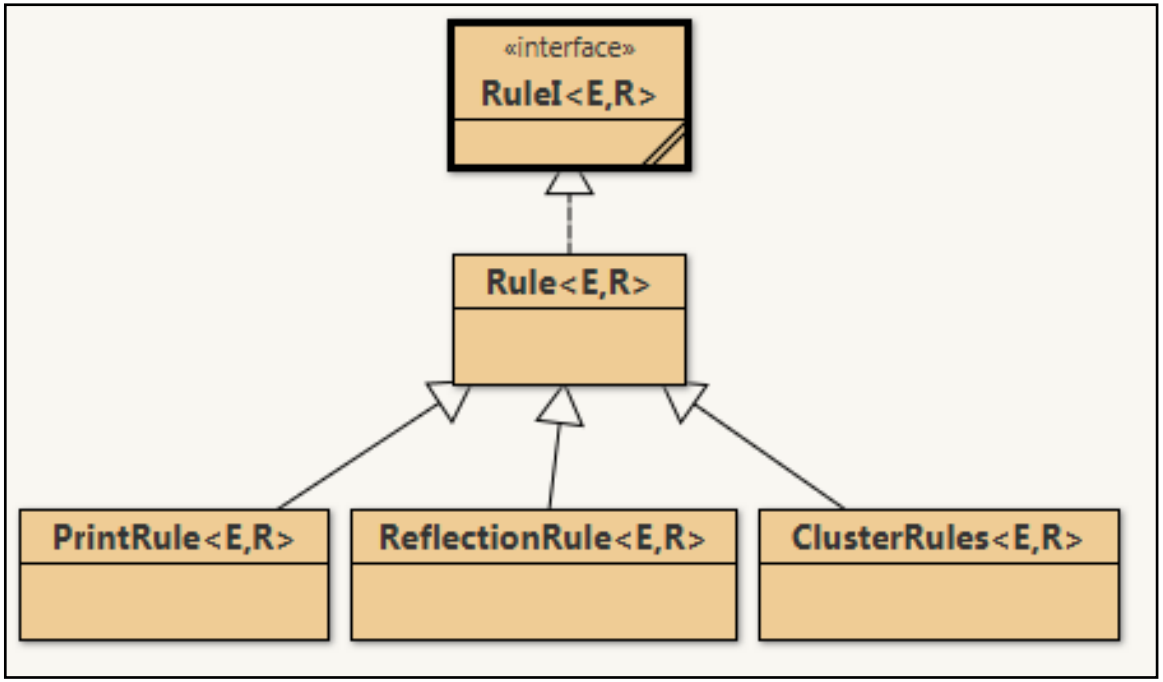
Un exemple simple, suite

```
public static class Inc extends Command<Integer>{
        public Integer execute(Integer i){
            return new Integer(i+1);
        }
}
```

- **Usage**

```
Command<Integer> inc = new Inc();
MacroCommand<Integer> plus2 = new MacroCommand<>();
plus2.add(inc).add(inc);

Integer res = new Integer(2);
res= plus2.execute(res);
assertEquals(4, res.intValue());
```

Les règles



- **Une règle est de la forme,**
  - **Si la specification est satisfaite alors la commande est exécutée**
  - **ClusterRules est le composite**

Règle, si condition alors commande

```
/** L'interface Règle.
 *  @param <E> La classe de l'entité sur laquelle porte la condition/specification
 *  @param <R> la classe de la donnée et du résultat
 */

public interface RuleI<E,R>{

    /**
     * Exécution d'une règle de type if condition alors exécution de la commande.
     * @param e l'entité sur laquelle porte la condition
     * @param r la donnée transmise
     * @return si la condition n'est pas satisfaite r est retourné,
     *         sinon le résultat de l'exécution de la commande est retourné
     */
     public R execute(E e,R r) throws Exception;
```

Règle, si condition alors commande

```
public  class Rule<E,R> implements RuleI<E,R>{
    protected Specification<E> specification;
    protected Command<R>       command;

    public Rule(Specification<E> specification, Command<R> command){
        this.specification = specification;
        this.command = command;
    }

    …
```

```
 public R execute(E e,R r) throws Exception {

      if(specification.isSatisfiedBy(e))
          return command.execute(r);

      return r;
 }
```

Un exemple simle

- **Usage**

```
Rule<Integer,Integer> rule = new Rule<>(new EstPair(), plus2);

Integer res = 0;
Integer x = 2;

res = rule.execute(x, res); // si x est pair alors res = res +2

assertEquals(2,res.intValue());
```

- **Une règle :**
- **Si *la spécification est satisfaite* alors *la commande* est exécutée.**

ClusterRules, le composite

```
class ClusterRules<E,R> extends Rule<E,R> implements Iterable<RuleI<E,R>>{
    private List<RuleI<E,R>> list;

    public ClusterRules(){
      this.list = new ArrayList<>();
    }

    public ClusterRules<E,R> add( RuleI<E,R> rule){
        list.add(rule);
        return this;
    }

    public Iterator<RuleI<E,R>> iterator(){
        return list.iterator();
    }

    public R execute(E e,R r) throws Exception {
        for(RuleI<E,R> rule:this){
            r = rule.execute(e, r);
        }
        return r;
    }

}
```

Un exemple simple, suite

```
Specification<Integer> inf = new EstInferieur(4);
MacroCommand<Integer> plus2 = new MacroCommand<Integer>();
plus2.add(new Inc()).add(new Inc());
Rule<Integer,Integer> rule = new Rule<>(inf,plus2);
```

- **Usage**

```
ClusterRules<Integer,Integer> cluster = new ClusterRules();
cluster.add(rule).add(rule).add(rule);

Integer res = cluster.execute(2,0);
assertEquals(new Integer(6),res);
```

Examen de janvier 2019

- **Démonstration/discussions**

Hypothèses et addition ?

- **Le pattern Specification**
  - **Autorise le regroupement de règles « au même endroit »**
    - **Une nouvelle règle engendre la modification cernée de l'un des composites**

- **femtoContainer, (un conteneur de beans correctement configurés)**
  - **Autorise l'injection de dépendances**

Le pattern Specification
\+ femtoContainer

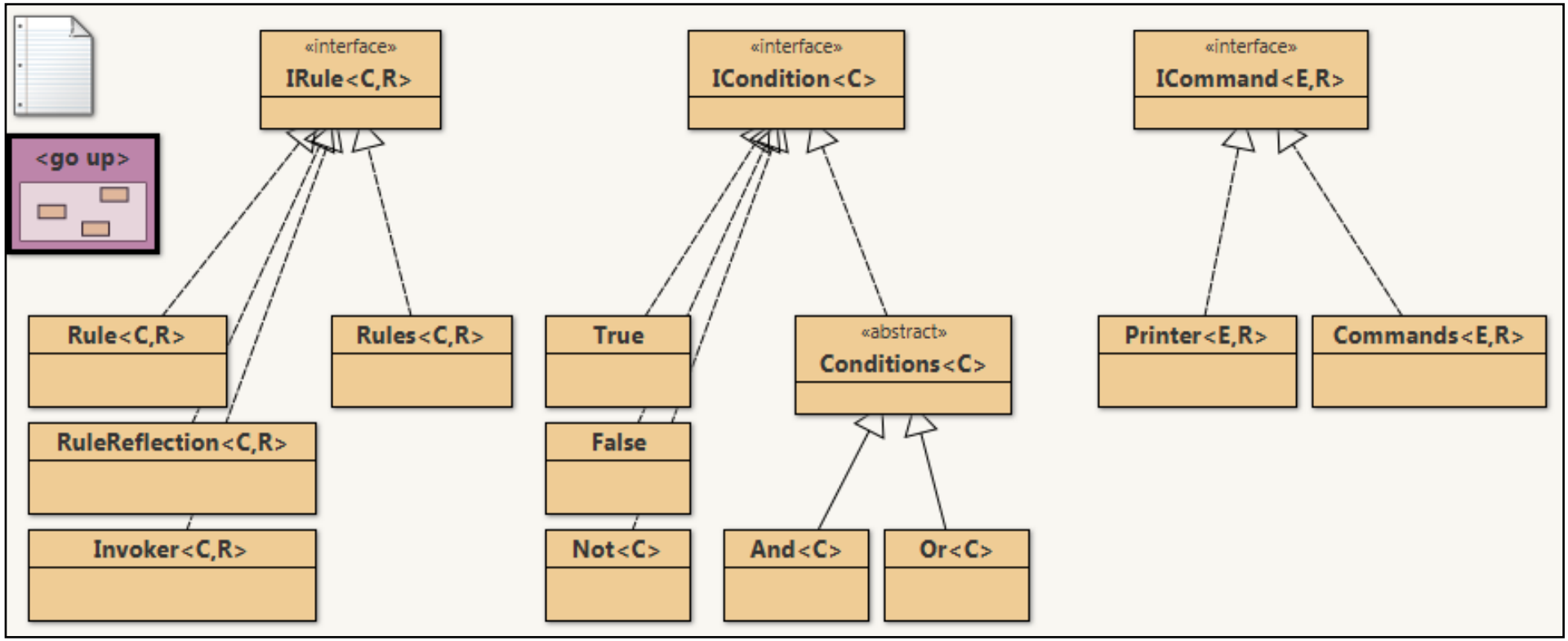
= VIP (Variability & Injection Pattern)

Sommaire VIP

Thème : Variabilité et injection$_{inject}$ de dépendances

- **Proposition: The VIP framework**
  - **Règle**
    - **Entité E, Résultat R**
    - **if (condition$_{inject}$ <E>){ commande$_{inject}$<E,R>}**
  - **Instruction**
    - **Contexte M, (une mémoire)**
    - **séquence<M>,**
      - **instruction1$_{inject}$<M> ; instruction2$_{inject}$<M>**
    - **sélection<M>,**
      - **if (condition$_{inject}$ <M>) instruction$_{inject}$<M> else instruction2$_{inject}$<M>**
    - **itération<M >**
      - **while (condition$_{inject}$ <M>) instruction$_{inject}$<M>**
  - **Règles, Commandes, Conditions**
  - **Invocateur, transaction**
  - **Règles et introspection**
  - **ServiceLocator…**
  - **…**

femtoContainer, paquetage vip



Outils prédéfinis : Java + un conteneur de beans

- **Java**
  - **Langage à Objets**
    - *Méthode sélectionnée dynamiquement en fonction de l'objet receveur*
  - **Généricité**
    - **List<Condition<E>>, List<Commande<E,R>>, Map<Agent,Resultat>, …**
  - **Introspection**
    - **Class.forName("CalculDesCongés"); "calculAncienneté".invoke(agent,résultat);**

- **Conteneur d'instances de classe : beans**
  - **Injection de dépendances**
    - **Injections d'instances créées**
    - **Injections décrites dans un fichier de configuration en texte**
  - **Couplage faible induit**
  - **Le code source Java ne change pas,**
    - **Seule la « configuration texte » évolue**

Objectifs

- **Quelles entités ?,** *issues du métier*
- **Pour quels types de résultats ?**
  - *Une valeur, une liste, un booléen, une table …*
  - *Effets de bord : affichage, requête REST, BD, HTTP …*
- **Une séquence de règles liées au métier :**
  - **si condition_1$_{inject}$(entité) alors commande_1$_{inject}$(entité,résultat)**
  - **si condition_2$_{inject}$(entité) alors commande _2$_{inject}$(entité,résultat)**
  - **si condition_3$_{inject}$(entité) alors commande _3$_{inject}$(entité,résultat)**
  - **si condition_4$_{inject}$(entité) alors commande _4$_{inject}$(entité,résultat)**
  - **…**

condition$_{inject}$ commande$_{inject}$ sont injectées

entité est immutable

résultat est mutable

Variabilité

- **Objectif**
  - Ajouter, retirer de nouvelles fonctionnalités
    - avec le moins d'impact possible
  - en utilisant une infrastructure *simple*
    - Une baie d'accueil des applications

***"In programming, simplicity and clarity are not a dispensable luxury, but a crucial matter that decides between success and failure."* - E. Dijkstra**

- **Afin de se concentrer**
  - **Sur les données et**
    les algorithmes, règles liés au métier
    . si condition(entité) alors commande(entité, résultat)

Proposition : une infrastructure simple

- **Commande : si condition alors commande**

En java une règle, ou commande élémentaire et générique

```
public class Rule<E,R> implements IRule<E,R>{

   public … execute(E entité,R résultat){

      if( condition.isSatisfied(entité) ){
        // alors
            command.execute(entité,résultat);
      }
   }

   ICondition<E>  condition;
   ICommand<E,R> command;
```

E comme Entité *(immutable)*
R comme Résultat *(mutable)*

Vers une infrastructure simple : une commande

- exécuter une **Règles : si condition alors commande**

```
public R execute(E entité,R résultat){

  if(condition.isSatisfied(entité)){

      return   command.executer(entité,résultat);

  }else
    return résultat

   ICondition<E>   condition;
   ICommande<E,R>  command;

}
```

condition *injectée*

commande *injectée*

Variabilité

```
bean.id.1=calcul
calcul.class=vip.rules.Rule
calcul.property.1=condition
calcul.property.1.param.1=estPair
calcul.property.2=command
calcul.property.2.param.1=plus2
```

Vers une infrastructure simple

- Une **commande** peut déclencher une exception

```
public class Rule<E,R> implements IRule<E,R>{

  public R execute(E entité,R resultat){

        if( condition.isSatisfied(entité) ){

            try{
                return
             command.execute(entité,resultat);

            }catch(VIPException e){
                exception.execute(entité,resultat);
             }
        }...
  }

  ICondition<E> condition;
  ICommand<E,R> command;
  ICommand<E,R> exception;
```

L'exception est une aussi *commande*

vers une infrastructure simple

```
public R executer(E entité,R résultat)throws RuntimeException{

  if(condition.isSatisfied(entité)){
    try{

      return command.execute(entité,résultat);

    }catch(VIPException e){

      exception.executer(entité,résultat);
    }
   }

   return résultat
  }

   ConditionI<E>  condition;
   CommandeI<E,R> command;
   CommandeI<E,R> exception;
}
```

condition *injectée*

commande *injectée*

exception *injectée*

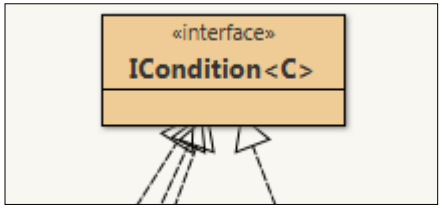
Variabilité

```
bean.id.1=calcul
calcul.class=vip.Rule
calcul.property.1=condition
calcul.property.1.param.1=estPair
calcul.property.2=command
calcul.property.2.param.1=plus2
calcul.property.3=exception
calcul.property.2.param.1=commandeException
```

Vers une infrastructure simple : la condition

si condition *alors commande*

- **Une condition élémentaire, générique**

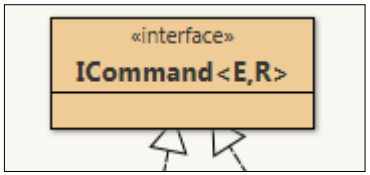



```
public interface IConditionI<E>{

    public boolean isSatisfied(E e);

}

E comme Entité
```

Vers une infrastructure simple: la commande

si condition alors commande

- **Une commande**




```
public interface ICommand<E,R>{

   public R execute(E entité, R résultat);

}
```

E comme Entité

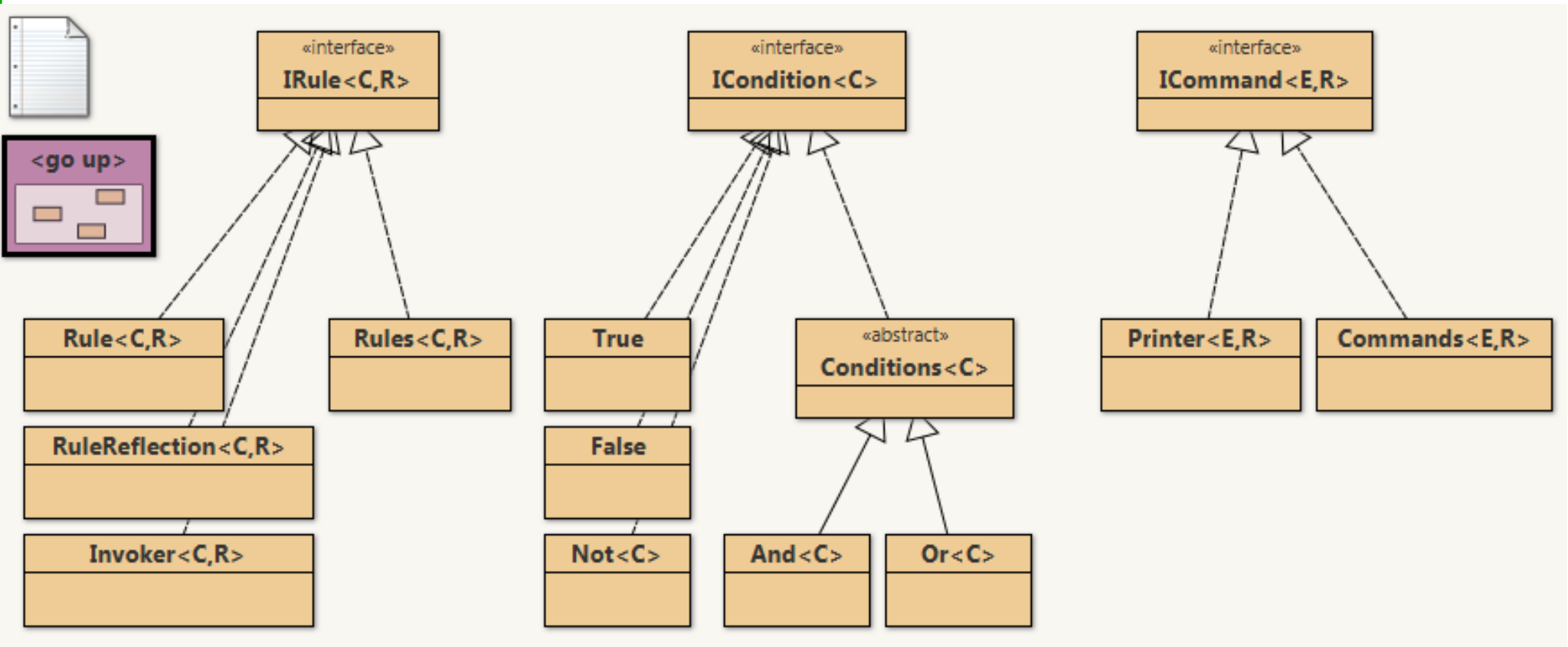
R comme Résultat

Initialisation d'une règle, comment ?

```
public class Rule<E,R> implements IRule<E,R>{
public R execute(E entité,R resultat)throws RuntimeException{
        if(condition.estSatisfaite(entity)){
            try{
                return commande.execute(entité,resultat);
            }catch(RuntimeException e){
                exception.execute(entité,resultat);
                throw e;
            }
        }
        return resultat;
    }
    ConditionI<E>  condition;  // ← injection
    CommandeI<E,R> command;    // ← injection
    CommandeI<E,R> exception;  // ← injection
    setCondition(….
}
```

- Injection par mutateurs via le fichier de configuration
  - Appels par le conteneur de *setCondition, setCommand, setException*

Paquetage vip.rules



- **En détail…**

Proposition :The VIP framework, prémisses

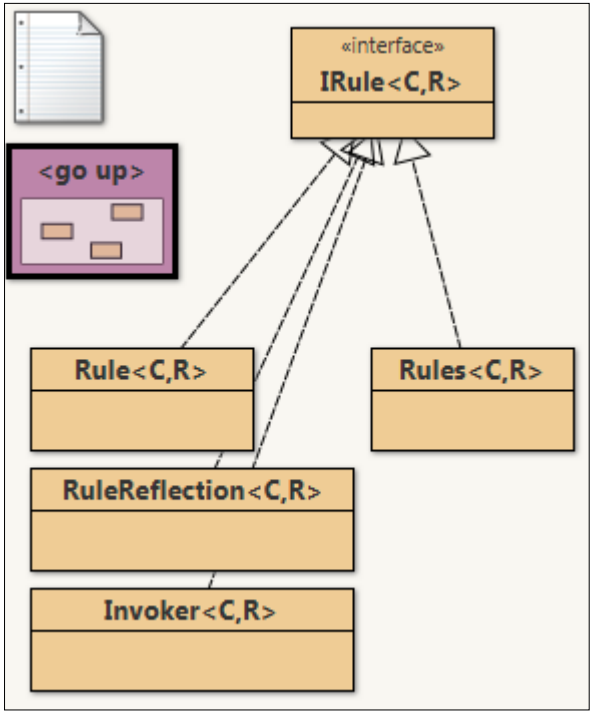
The VIP language

- **Un langage simple**
  - **Règle**
    - **Si Condition alors Commande**

VIP commme Variability & Injection Pattern

Invocateur (cf. patron Commande…)

- **L'invocateur de règle**
- **L'invocateur est une règle**



- **La Règle devient « transactionnelle »**
  - **L'invocateur autorise un retour arrière**
    - **Apparenté commit/rollback**
    - **Restitution de la valeur résultat** (mutable) **avant son appel**

Variabilité de la commande

Un invocateur quelque soit la règle

```
public class Invoker<C,R extends Serializable> implements IRule<C,R>{

    private IRule<C,R>   rule; // <- injection
    private Stack<byte[]> stk;

    public Invoker(){this.stk = new Stack<byte[]>();}
    public void setRule(IRule<C,R> rule){this.rule = rule; }

    public R execute(C context,R result){
        save(result);
        return rule.execute(context,result);
    }
    public R undo(){
        return restore();
    }


}
```

Résumé d'étape

- **Nous avons les instructions élémentaires** (*injectées*)
  - **Invocateur, Commande,**

```
si Condition
alors Commande
```

```
try
    si Condition
    alors Commande
catch( e)
    exception
```

  - **Plusieurs Commandes ?**
  - **Plusieurs Conditions**
  - **Et les Instructions ?**

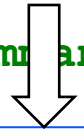
Résumé d'étape, suite

- **Plusieurs Commandes ?**
  - Une table de commandes
- **Plusieurs Conditions ?**
  - ET, OU, NON, Il existe, Quelque soit, Aucune ?
- **Plusieurs Règles ?**
  - Une table de règles

  - Commande composite,
    - Commands
      - commande1 commande2 commande3 …
  - Condition composite
    - Conditions
      - condition1 condition2 condition3 …
  - Règle composite
    - Rules
      - rule1 rule2 rule3 …

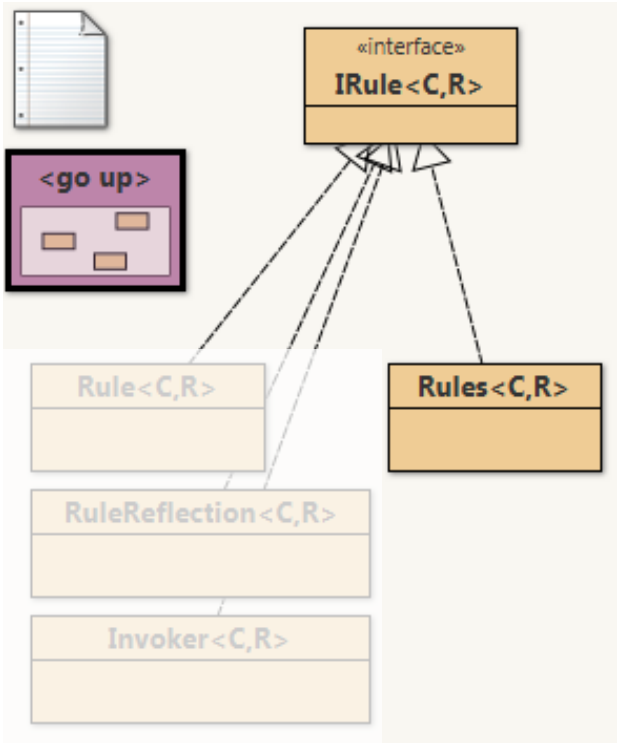
MacroCommande, ou règle composite

```
public class Commands<E,R> implements ICommand<E,R>{

  public R executer(final E entity,final R result){

        for(ICommand<E,R> cmd : commands){
            result = cmd.executer(entite, result);
        }
        return result

    }

 private ICommand<E,R>[] commands; // <- injection
```



```
résultat=commandes[1].executer (Entité, résultat)
résultat=commandes[2].executer (entité, résultat)
résultat=commandes[3].executer (entité, résultat)
résultat=commandes[4].executer (entité, résultat)
résultat=commandes[5].executer (entité, résultat)
…
```

Règles, un composite



- **Rules**
  - **IRuleI<E,R>[] une table de commandel**
  - **Structure de données récursives**

A nouveau rappel de l'objectif

- **Objectif**
  - ***Ajouter, retirer de nouvelles fonctionnalités***
    - ***avec le moins d'impact possible***
  - ***Proposer une infrastructure*** *simple*
    - ***Une baie d'accueil (un framework et un langage)***

- **Afin de se concentrer**
  - **Sur les données et**
    les algorithmes, règles liés au métier
    si condition alors commande

Résumé : le VIP-langage

- **Nous avons les instructions élémentaires**

```
si Condition
alors Commande
```

```
try
  si Condition
  alors Opération
catch
  exception
```

- **Instructions composites**
  - Commands
    - ICommand[]
  - Conditions
    - IConditions[]
  - Rules
    - IRule[]

La démarche résumée : 1) « spécification »

- **Quelles entités ?,** *issues du métier*
  - *Auditeur, Cnam, Service, …*
- **Pour quels types de résultats ?**
  - ***Une valeur, une liste, un booléen, Requêtes web, …***
- **Une séquence de règles :**
  - **si conditions_1(Entité) alors operations_1(Entité, résultat[1])**
  - **si conditions_2(Entité) alors operations_2(Entité, résultat[2])**
  - **si conditions_3(Entité) alors operations_3(Entité, résultat[3])**
  - **si conditions_4(Entité) alors operations_4(Entité, résultat[4])**
  - **si conditions_5(Entité) alors operations_5(Entité, résultat[5])**
  - **…**

La démarche suite, 2) conception

si conditions_1(E) alors commande_1(E,R)

- **conditions_1 les conditions,** tests sur l'entité E

devient

public class Conditions_1 implements ICondition<E>{

boolean estSatisfaite(E entité){

***return* conditions_1(E)** ⟺ *liée au métier*

}

- **commande_1 les opérations,** tests sur l'entité E, pour un résultat R

devient

public class Commande_1 implements ICommande<E,R>{

R execute (E entité, R résultat){

***return* un calcul(E,R)** ⟺ *liée au métier*

}

Règle par introspection

Comment déclencher une méthode

si c'est une nouvelle entité ?, …

- **Une nouvelle entité → une nouvelle classe**
  - **Avec de nouvelles méthodes**

Une idée …

- **Les noms des méthodes sont injectées**
  - **RuleReflection est une règle**
  - **Les noms pour la condition, la commande et l'exception sont injectées**

RègleParIntrospection.java, un extrait…

```
public class RuleReflection<C,R> implements IRule<C,R>{
    protected ICondition<?> instanceCondition=null;
    protected ICommand<?,?> instanceCommand=null;
    public void setCondition(String condition){
        try{
          Class<?> classCondition = Class.forName(condition);
          instanceCondition = (ICondition<?>)classCondition.newInstance();
        }catch(Exception e){}
    }
    public ICondition<?> getCondition(){ return instanceCondition;}
    public void setCommand(String command){
        try{ Class<?> classCommand = Class.forName(command);
             instanceCommand = (ICommand<?,?>) classCommand.newInstance();
        }catch(Exception e){}
    }
    public ICommand<?,?> getCommand(){ return instanceCommand;}

   public R execute(C context,R result){
```

Diapositive suivante

```
   }
```

CommandeParIntrospection.java, un extrait…

```
public class RuleReflection<C,R> implements IRule<C,R>{
    protected ICondition<?> instanceCondition=null;
    protected ICommand<?,?> instanceCommand=null;

public R execute(C context,R result){
     Method mCondition=null, mCommand=null;
     try{
      mCondition =
    instanceCondition.getClass().getDeclaredMethod("isSatisfied",context.getClass());

      mCommand = instanceCommand.getClass().getDeclaredMethod("execute",result.getClass());

       condition = (boolean)mCondition.invoke(instanceCondition,context);

    if(condition)
      return (R)mCommand.invoke(instanceCommand,context,result);
    }catch(Exception e){
       throw new RuntimeException(e.getMessage());
    }
    return result;
```

Découverte dynamique du type de l'entité

Configuration, un extrait

```
bean.id.1=invocateur
invocateur.class=vip.rules.Invoker
invocateur.property.1=rule
invocateur.property.1.param.1=rule

bean.id.2=rule
rule.class=vip.rules.RuleReflection
rule.property.1=condition
rule.property.1.param.1=EstPair
rule.property.2=command
rule.property.1.param.2=Inc
```

Démonstration, énoncé janvier 2019 avec VIP

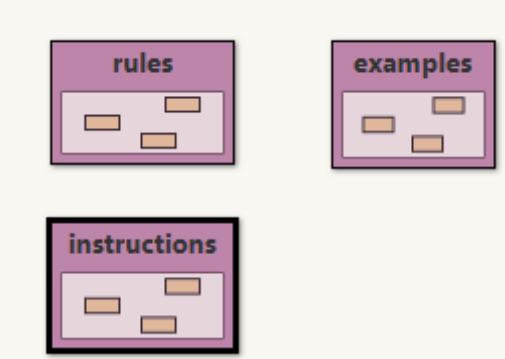
Résumé d'étape

"In programming, simplicity and clarity are not a dispensable luxury, but a crucial matter that decides between success and failure." - E. Dijkstra

- **The VIP framework**
  - **Un langage simple**
    - **Règle,**
      - **si condition alors operation**
    - **Rules, Commands, Conditions**
      - **Une séquence de règles, de commandes, de conditions**
  - **Une configuration simple**
    - **Un fichier texte**
  - **VIP Variability & Injection Pattern**
    - **sans impact sur le code**

The VIP framework

- **The VIP framework**
- **Un langage simple**
  - Rule
    - Si Condition alors Commande
  - Instructions



- Rule
- Sequence
- Selection
- TantQue

Injection

Configuration.texte

VIP commme Variability & Injection Pattern

The VIP framework/language

- The VIP
- Proposition pour discussion

  - Un langage simple “injectéé
    - Une séquence de règles, de conditions, de commandes

    - Instruction
      - Sequence
        instruction1 ; instruction2
      - Sélection
        si condition alors instruction1 sinon instruction2
      - TantQue
        tantQue condition faire instruction

Instructions injectées

- **Instruction**
  - **Sequence**

    instruction1

    instruction2

    <- instructions injectées

  - **Sélection**

    si condition

    <- condition injectée

    alors instruction1

    sinon instruction2

    <- injection

    <- injection

  - **TantQue**

    tantQue

    condition

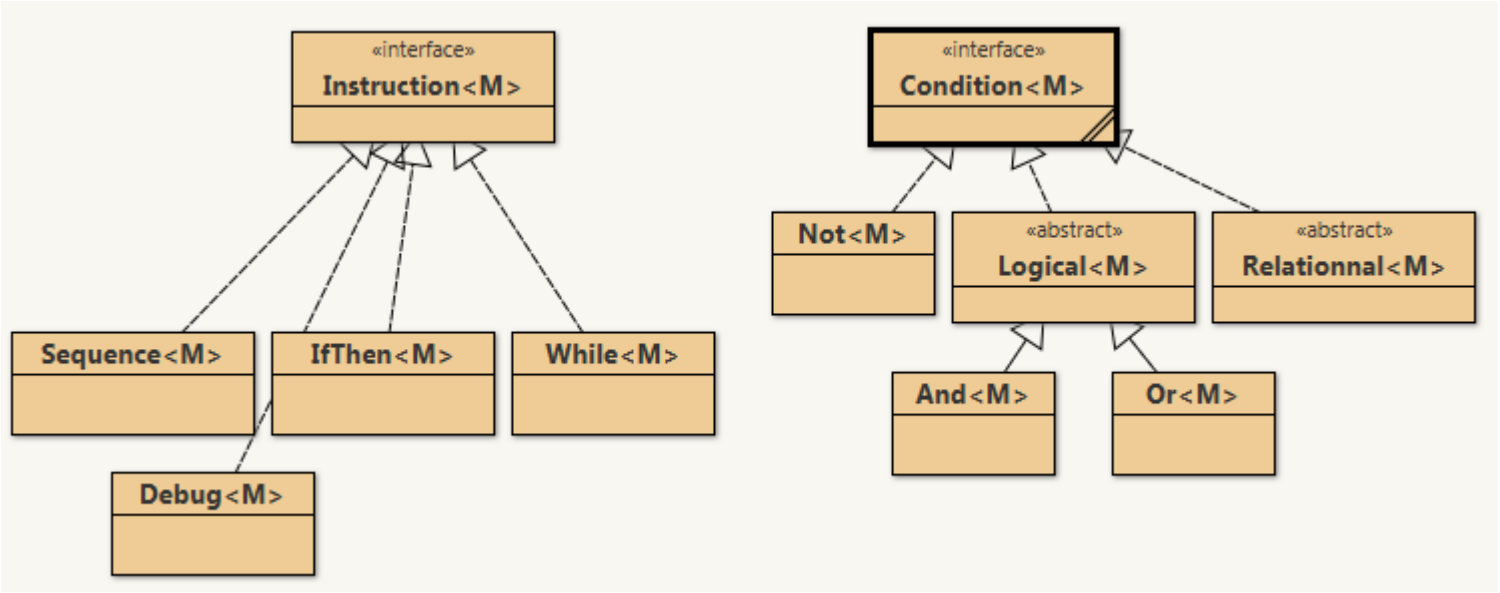
    faire

    instruction

    <- injection

    <- injection

VIP : les instructions, paquetage vip.instructions



- **M comme mémoire mutable**
- **Une instruction**
  - En général, effectue un changement d'état de la mémoire

```
public interface Instruction<M>{
  public M execute(M memory);
}
```

  - **Sequence, Selection, TantQue,**

Sequence<M>

```
public class Sequence<M> implements Instruction<M>{
  protected Instruction<M>  instruction1; // ← injection
  protected Instruction<M>  instruction2; // ← injection

  public void setInstruction1(Instruction<M> instruction1){
      this.instruction1 = instruction1;
  }
  public void setInstruction2(Instruction<M> instruction2){
      this.instruction2 = instruction2;
  }

  public M execute(M memory){
    M memory1 = instruction1.execute(memory);

    M memory2 = instruction2.execute(memory1);

    return memory2;
    }
}
```

Sélection<C,E,R>

```
public class IfThen<M> implements Instruction<M>{
  protected Condition<M>    condition;     // ← injection
  protected Instruction<M>  instruction;   // ← injection

  public void setCondition(Condition<M> condition){
      this.condition = condition;
  }
  public void setInstruction(Instruction<M> instruction){
      this.instruction = instruction;
  }
  public M execute(M memory){
      assert condition!=null && instruction!=null;

      if (condition.isSatisfied(memory)){
         M m1 = instruction.execute(memory);

         return m1;
      }else
         return memory;
    }
}
```

TantQue<C,E,R>

```
public class While<M> implements Instruction<M>{

  protected Condition<M>    condition;          <- injection
  protected Instruction<M>  instruction;        <- injection
…

  public M execute(M memory){
      assert condition!=null && instruction!=null;

      if(condition.isSatisfied(memory)){
          M m1 = instruction.execute(memory);
          while (condition.isSatisfied(m1)){
               m1 = instruction.execute(m1);

          }
           return m1;
      }else
        return memory;
      }
}
```

Un exemple/démonstration

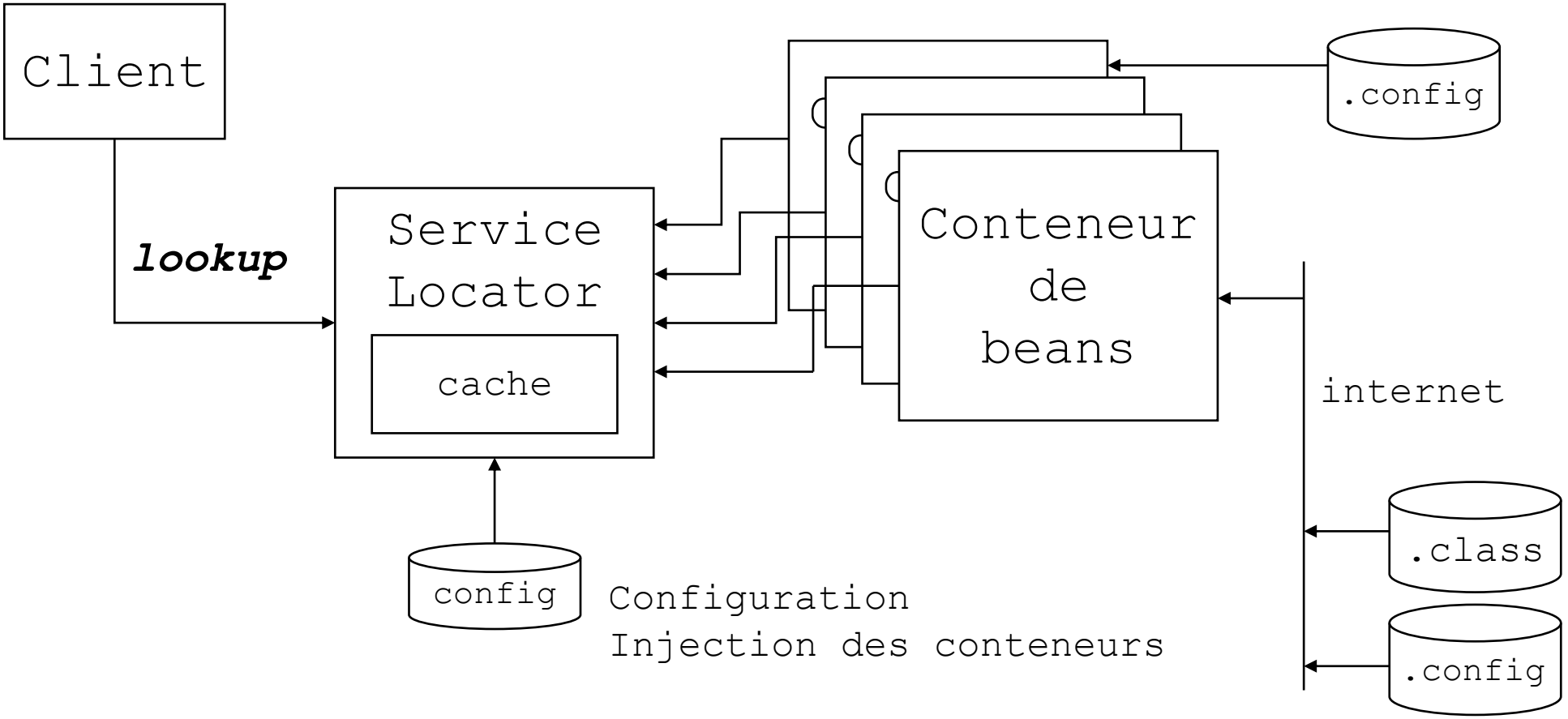
```
bean.id.2=inc
inc.class=vip.examples.instructions.Inc
inc.property.1=operand
inc.property.1.param.1=2

bean.id.3=inf
inf.class=vip.examples.instructions.LessThan
inf.property.1=operand
inf.property.1.param.1=10

bean.id.7=while
while.class=vip.instructions.While
while.property.1=condition
while.property.1.param.1=inf
while.property.2=instruction
while.property.2.param.1=inc
```

Résumé, discussions

Factorisation de conteneurs

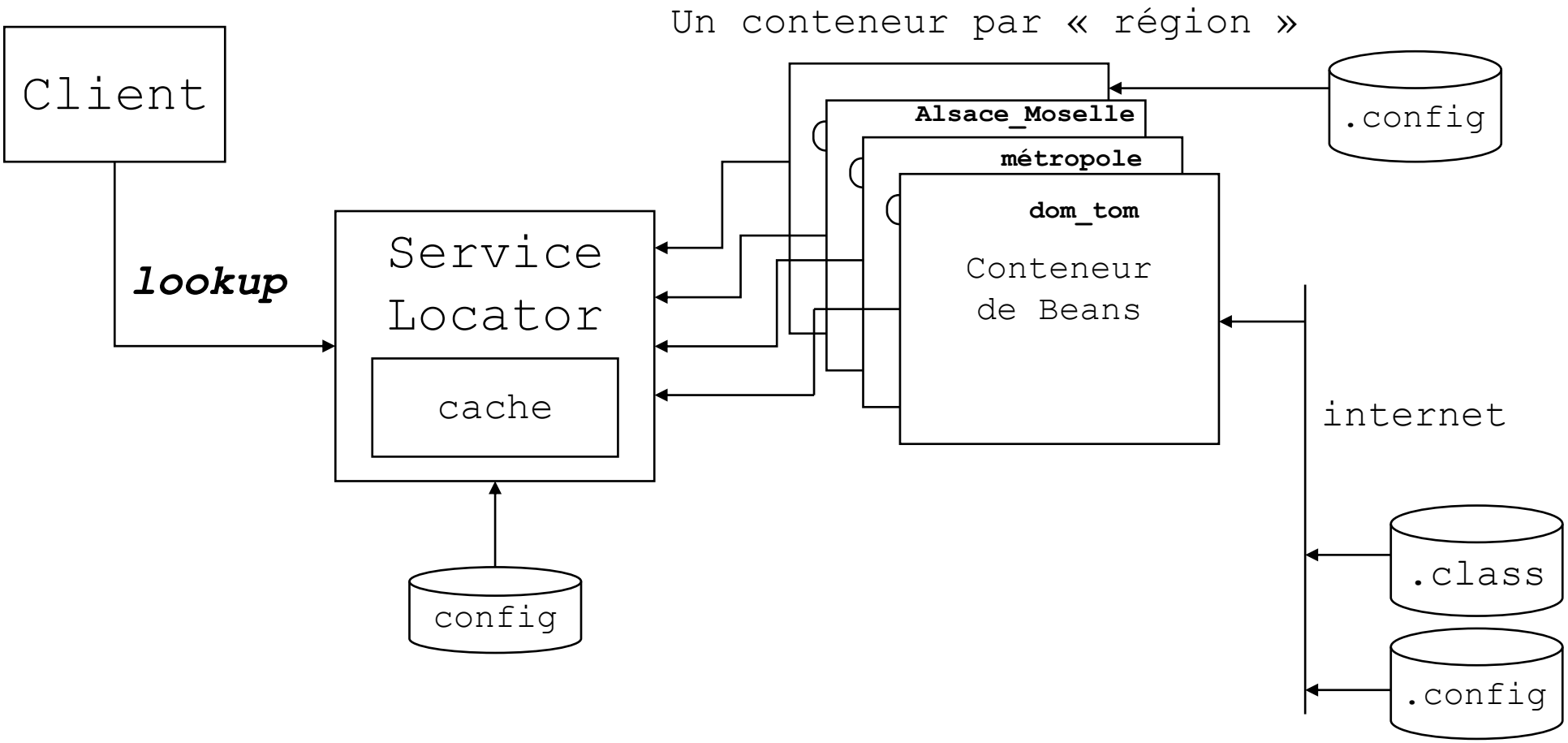
- **Espace de beans,**
- **Conteneur de conteneurs**
- **ServiceLocator**

Un service locator adapté… un conteneur de conteneur



- **Ici, une agrégation de conteneurs**
  - **Chaque conteneur est correctement configuré**
- **Le client demande un service, peu importe d'où il vient…**

Un service locator adapté…



- **Ici, un conteneur, une configuration par région**
- **The VIP framework, (le patron ServiceLocator est inclus)**

Discussion, conclusion, perspectives

- **The VIP framework**
  - **Un langage simple**
    - **3 instructions élémentaires**
      - **Condition, Command, Rule**
    - **3 instructions composites**
      - **Conditions, Command, Rules**
    - **Instructions**
      - **Séquence, sélection et itération**
  - **Une configuration**
    - **un fichier texte**
  - **-> Variabilité effective**
  - **génération automatique de la configuration possible**

Discussion

- **Solution VIP framework**
  - Simple
  - Immédiate
  - Pas de formation
  - Pas d'inconnue technique
- **Moteur de règles**
  - Drools, Jess, Rulebook, Easy rules …
    - Apprentissage nécessaire
    - Solution éprouvée
- **Variabilité : Moteur de règles + Spring ?**

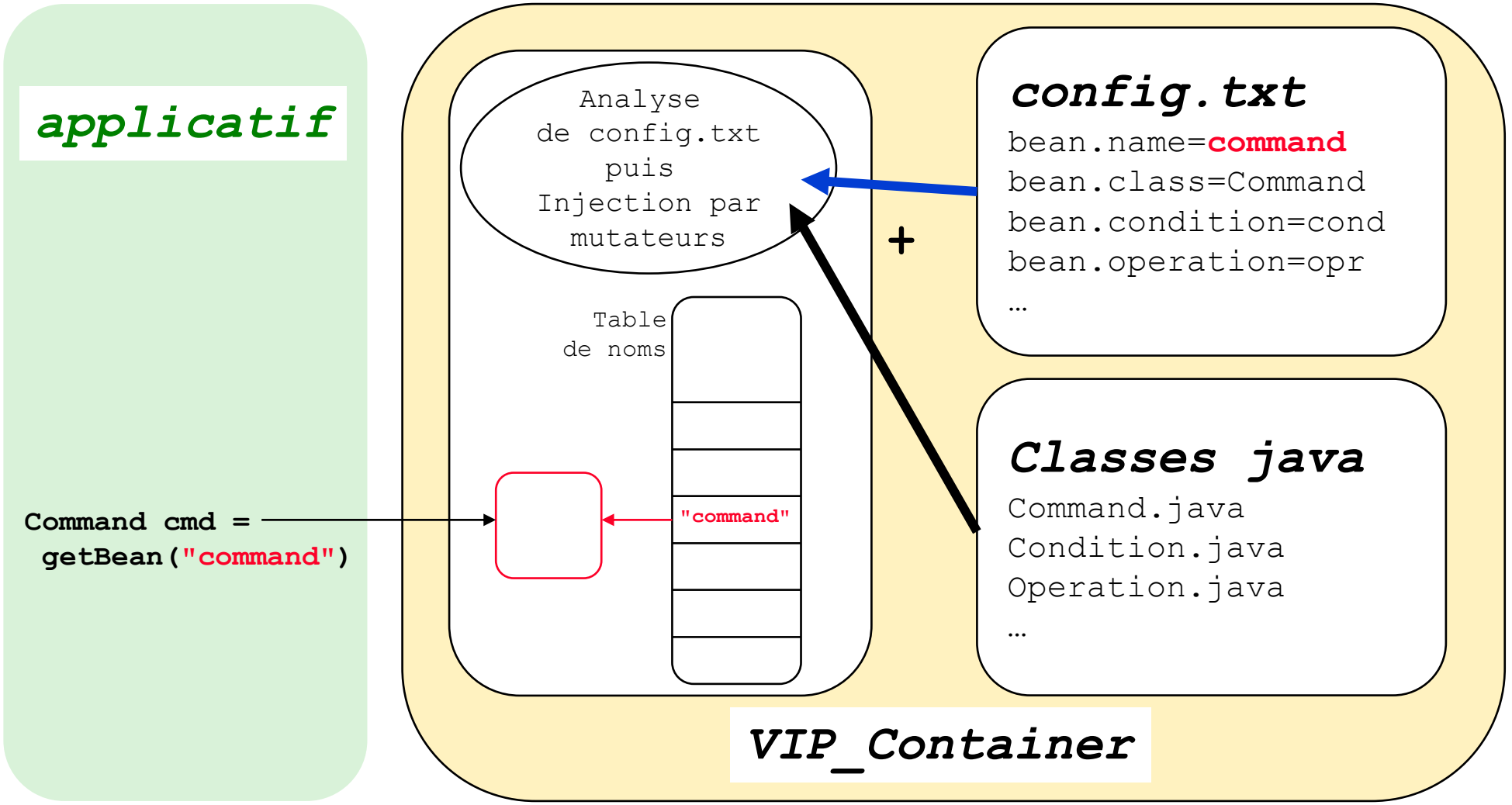
Annexes

- **Le conteneur de beans**
  - **Description et fonctionnement en 4 diapositives**

Outil d'injection : une explication sommaire

- **femtoContainer**
  - **Un outil d'injection nécessité des mutateurs**
  - **Description sommaire du fonctionnement**

VIP Container



- **Injection par mutateur**
  - **Usage de l'introspection**
- **http://jfod.cnam.fr/progAvancee/femto_container_distrib_bluej.jar**

Outil d'injection : une explication sommaire

L'analyse de ce texte*

```
bean.id.2=calculConge
calculConge.class=injection_decorateur.CalculConges
calculConge.property.1=calculette
calculConge.property.1.param.1=calculCongeBonifie
```

Engendre au sein du conteneur

```
calculConge = new injection_decorateur.CalculConges();
calculConge.setCalculette(calculCongeBonifie);
```

L'*utilisateur* demande une référence à l'aide d'un nom

```
ApplicationContext ctx = Factory.createApplicationContext();
 CongesI conges = (CongesI) ctx.getBean("calculConge");
  Contexte contexte = …
  Agent paul = …

int nombreDeJoursDeCongés = conges.calculer(contexte, paul);
```

Bean : https://fr.wikipedia.org/wiki/JavaBeans

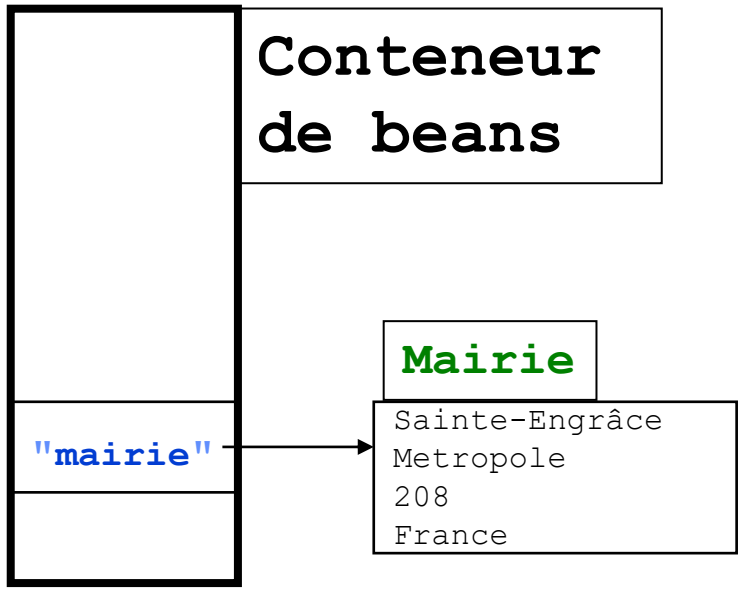
* Ici un fichier de propriétés en java cf. java.util.Properties, Autres formtas possibles XML, JSON …

La configuration : syntaxe

- **Java :Une classe et des attributs (propriétés)**

```
public class Mairie extends Contexte{
  private String nom;
  private String lieu;
  private String pays;
  private int nombreHabitant
  public Mairie(){}
  public void setNom(String nom){this.nom=nom;}
  public void setLieu(String lieu){this.lieu=lieu;}
  …
}
```

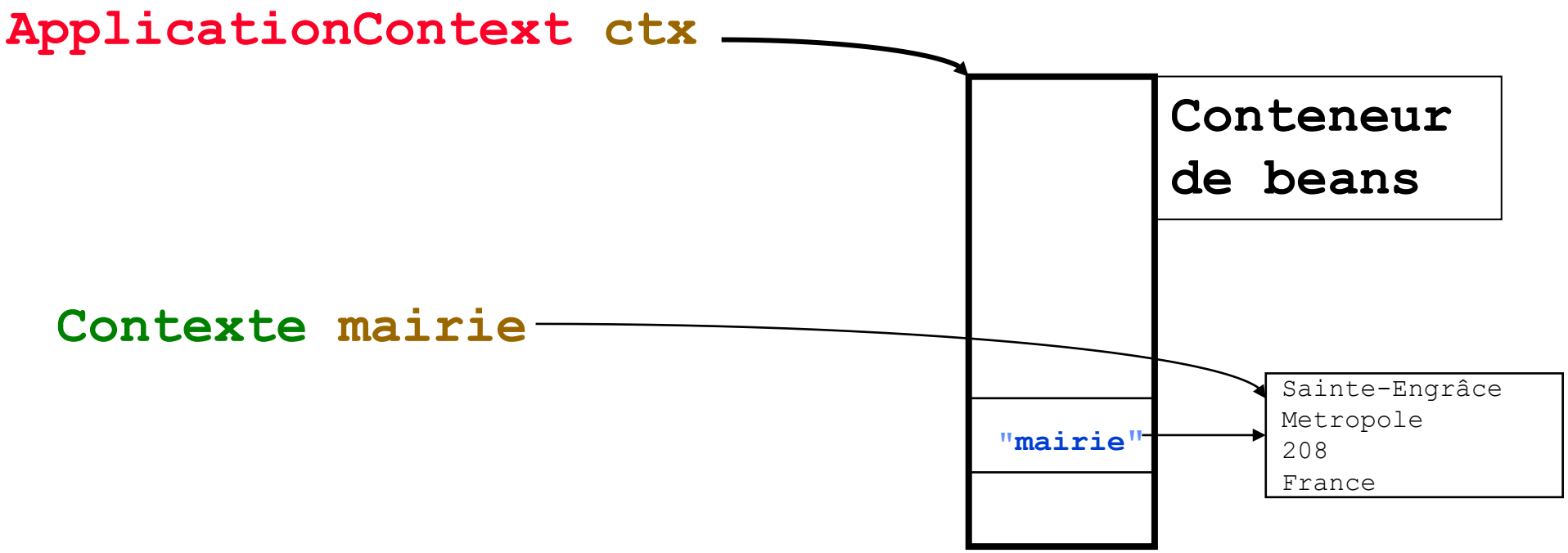
```
# le nom du « bean »
bean.id.1=mairie
# la classe Java
mairie.class=Mairie
# les attributs
mairie.property.1=nom
mairie.property.1.param.1=Sainte-Engrâce
mairie.property.2=lieu
mairie.property.2.param.1=Metropole
mairie.property.3=nombreHabitants
mairie.property.3.param.1=208
mairie.property.4=pays
mairie.property.4.param.1=France
```



Utilisation: syntaxe

- **Java**
  - Utilisation de femtoContainer

```
ApplicationContext ctx = Factory.createApplicationContext();
Contexte mairie = (Mairie) ctx.getBean("mairie");
```



Configuration en XML

Configuration, ici un fichier xml, syntaxe apparentée Spring

```
<beans>
  <bean> id=calculParDefaut
         class=injection_decorateur.CalculParDefaut
  </bean>

  <bean> id=calculConge
         class=injection_decorateur.CalculConges>
    <property name=calculette>
    <ref local=calculCongeBonifie/>
 </bean>

 <bean> id=calculCongeBonifie
        class=injection_decorateur.CalculConges>
    <property name=calculette>
    <ref local=calculParDefaut/>
 </bean>

……
</beans>
```